Chapter 15

Neural Network Models

A major recent development in statistical machine translation is the adoption of neural networks to model components of a system, and even build fully integrated end-to-end systems. The promise of neural network models is to better share statistical evidence between similar words and more flexible handling of rich context. This chapter introduces several neural network modeling techniques and how they are applied to problems in machine translation

15.1 Introduction to Neural Networks

A neural network is a machine learning technique that takes a number of inputs and predicts the output. In a way, they are not very different from linear models but overcome some of their short-comings.

15.1.1 Linear Models

We already encountered previously linear models that use a number of feature values and their weights to assign scores to a given instance. Recall Equation **??**, where a potential translation x of a sentence was represented by a set of features $h_i(x)$. Each feature is weighted by a parameter λ_i to obtain the overall score. Ignoring the exponential function that we used previously to turn the linear model into a log-linear model, we have now

$$\text{score}(\lambda, x) = \sum_j \lambda_j \; h_j(x) \tag{15.1}$$

Graphically, a linear model can be illustrated by a network, where feature values are input nodes, arrows are weights, and the score is an output node (see Figure 15.1).

Most prominently, we use linear models to combine different components of a machine translation system, such as the language model, the phrase translation model, the reordering model, but also properties such as the length of the sentence, or the accumulated jump distance between phrase translations. We presented training methods to assign weights λ_i to each of these features $h_i(x)$, related to their importance in contributing to obtaining a good translation.

However, these linear models do not allow us to define more complex relationships between the features. Let us say that we find that for short sentences the language model is less important

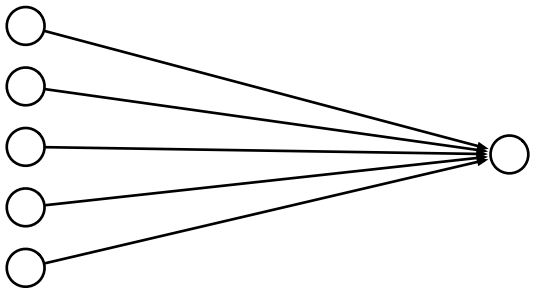


Figure 15.1: Graphical illustration of a linear model as a network: feature values are input nodes, arrows are weights, and the score is an output node.

than the translation model, or that average phrase translation probabilities higher than 0.1 are all very similar and reasonable but any value below that is really terrible. The first hypothetical example is an implies dependence between features and the second example implies non-linear relationship between feature value and impact on the final score.

A commonly cited counter-example to the use of linear models is XOR, i.e., the boolean operator $\oplus$ with the truth table $0 \oplus 0 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$, and $1 \oplus 1 = 0$. For a linear model with two features (representing the inputs), it is not possible to come up with weights that give the correct output in all cases. Linear models assume that all instances, represented as points in the feature space, are linearly separable. This is not the case with XOR, and may not be the case for type of things we want to model in machine translation.

15.1.2 Multiple Layers

Neural networks modify linear models in two important ways. The first is the use of multiple layers. Instead of computing the output value directly from the input values, a **hidden layer** is introduced. It is called hidden, because we can observe inputs and outputs in training instances, but not the mechanism that connects them — this use of the concept hidden is similar to hidden Markov models.

See Figure 15.2 for on illustration. The network is processed in two steps. First, the value of each hidden node is computed based on the linear combination of weighted input node values. Then, the value of the output node is computed based on the linear combination of weighted hidden node values.

At this point, let us change the mathematical notations to be more consistent with the neural network literature. A neural network with a hidden layer consists of

- a vector of input nodes with values $\vec{x} = (x_1, x_2, x_3, ...x_n)^T$
- a vector of hidden nodes with values $\vec{h} = (h_1, h_2, h_3, ...h_m)^T$
- a vector of output nodes with values $\vec{y} = (y_1, y_2, y_3, ...y_l)^T$
- a matrix of weights connecting input nodes with hidden nodes W
- a matrix of weights connecting hidden nodes with output nodes U

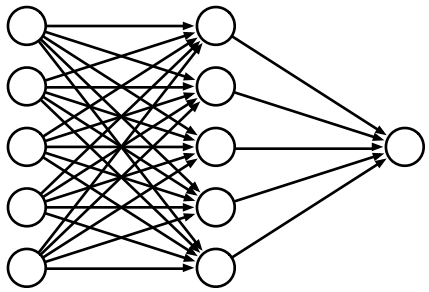


Figure 15.2: A neural network with a hidden layer.

The computations in a neural network with a hidden layer, as sketched out so far, are

$$h_j = \sum_i x_i w_{ji} \tag{15.2}$$

$$y_k = \sum_j h_j u_{kj} \tag{15.3}$$

Note that we snuck in the possibility of multiple output nodes y_k, although our figures so far only showed one.

15.1.3 Non-Linearity

If we carefully think about the addition of a hidden layer, we realize that we have not gained anything so far in terms of power to model input/output relationships. We can easily do away with the hidden layer by multiplying out the weights

$$\begin{aligned} y_k &= \sum_j h_j u_{kj} \\ &= \sum_j \sum_i x_i w_{ji} u_{kj} \\ &= \sum_i x_i \left(\sum_j u_{kj} w_{ji} \right) \end{aligned} \tag{15.4}$$

Hence, a salient element of neural networks is the use of a **non-linear activation function**. After computing the linear combination of weighted feature values $s_j = \sum_i x_i w_{ji}$, we obtain the value of a node only after applying this function $h_j = f(s_j)$.

Popular choices are the **hyperbolic tangent** tanh(x) and the **logistic function** sigmoid(x). See Figure 15.3 for more details on these functions. A good way to think about these activation functions is that they segment the range of values for the linear combination s_j into

- a segment where the node is turned off (values close to 0 for tanh, or –1 for sigmoid)
- a transition segment where the node is partly turned on
- a segment where the node is turned on (values close to 1)

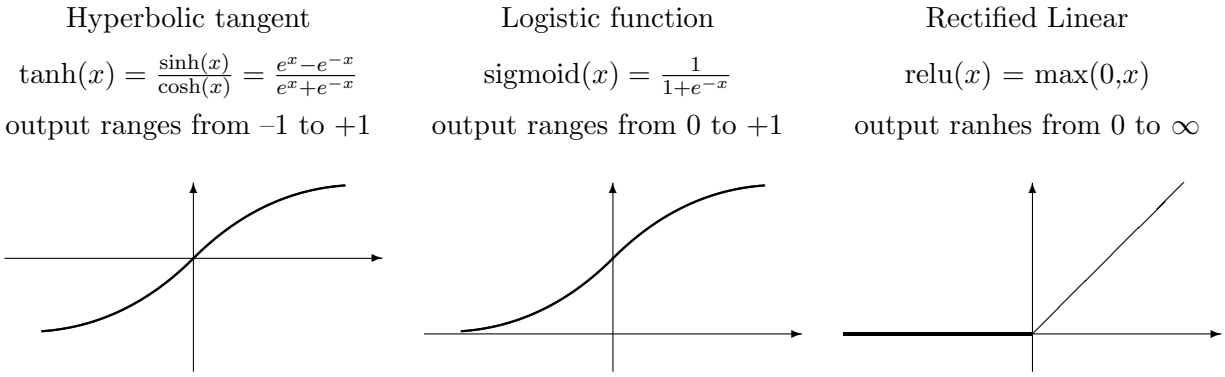


Figure 15.3: Typical activation functions in neural networks.

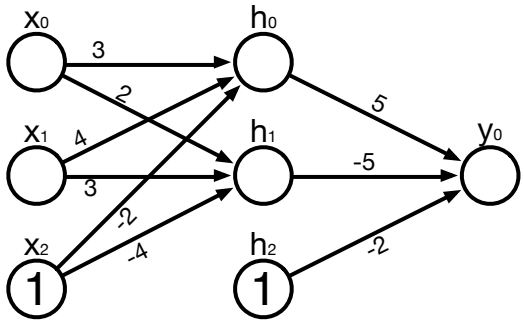


Figure 15.4: A simple neural network with bias nodes in input and hidden layers.

A different popular choice is the activation function for the **rectified linear unit** (ReLU). It does not allow for negative values and floors them at 0, but does not alter the value of positive values. It is similar to the activation functions above but faster to compute.

You could view each hidden node as a feature detector. For a certain configurations of input node values, it is turned on, for others it is turned off. Advocates of neural networks claim that the use of hidden nodes obviates (or at least drastically reduces) the need for feature engineering: Instead of manually detecting useful patterns in input values, the training of the hidden nodes discovers them automatically.

We do not have to stop at a single hidden layer. The currently fashionable name **deep learning** for neural networks stems from the fact that often better performance can be achieved by deeply stacking together layers and layers of hidden nodes.

15.1.4 Inference

Let us walk through neural network inference (i.e., how output values are computed from input values) with a concrete example. Consider the neural network in Figure 15.4. This network has one additional innovation that we have not presented so far: bias units. These are nodes that always have the value 1. Such bias units give the network something to work with in the case that all input values are 0, and hence the weighted sum s_j is 0 not matter the weights.

Layer	Node	Summation	Activation
hidden	h_0	$1.00 \times 3.00 + 0.00 \times 4.00 + 1.00 \times -2.00 = 1.00$	0.73
hidden	h_1	$1.00 \times 2.00 + 0.00 \times 3.00 + 1.00 \times -4.00 = -2.00$	0.12
output	y_0	$0.73 \times 5.00 + 0.12 \times -5.00 + 1.00 \times -2.00 = 1.06$	0.74

Table 15.1: Calculations for input (1,0) to the neural network in Figure 15.4.

Let us use this neural network to process some input, say the value 1 for the first input node x_0 and 0 for the second input node x_1. The value of the bias input node (labelled x_2) is fixed to 1. To compute the value of the first hidden node h_0, we have to carry out the following calculation.

$$\begin{aligned} h_0 &= \text{sigmoid}\left(\sum_i x_i w_{1i}\right) \\ &= \text{sigmoid}\left(1 \times 3 + 0 \times 4 + 1 \times -2\right) \\ &= \text{sigmoid}\left(1\right) \\ &= 0.73 \end{aligned} \tag{15.5}$$

The calculations for the other nodes are summarized in Table 15.1. The output value in node y_0 for the input (0,1) is 0.74. If we expect binary output, we would understand this result as the value 1, since it is over the threshold of 0.5 in the range of possible output values [0;1].

Here, the output for all possible binary inputs:

Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.12	0.02	$0.18 \rightarrow 0$
0	1	0.88	0.27	$0.74 \rightarrow 1$
1	0	0.73	0.12	$0.74 \rightarrow 1$
1	1	0.99	0.73	$0.33 \rightarrow 0$

Our neural network computes XOR. How does it do that? If we look at the hidden nodes h_0 and h_1, we notice that h_0 acts like the Boolean OR: Its value is high if at least of the two input values is 1 ($h_0 = 0.88$, 0.73, 0.99, for the three configurations), other it has a low value (0.12). The other hidden node h_1 acts like the Boolean AND - it only has a high value (0.73) if both inputs are 1. XOR is effectively implemented as the subtraction of the AND from the OR hidden node.

Note that the non-linearity is key here. Since the value for the OR node h_0 is not that much higher for the input of (1,1) opposed to one one 1 in the input (0.99 vs. 0.88 and 0.73), the distinct high value for the AND node h_1 in this case (0.73) manages to push the final output y_0 below the threshold. This would not be possible if the impact of the inputs would be simply summed up as in linear models.

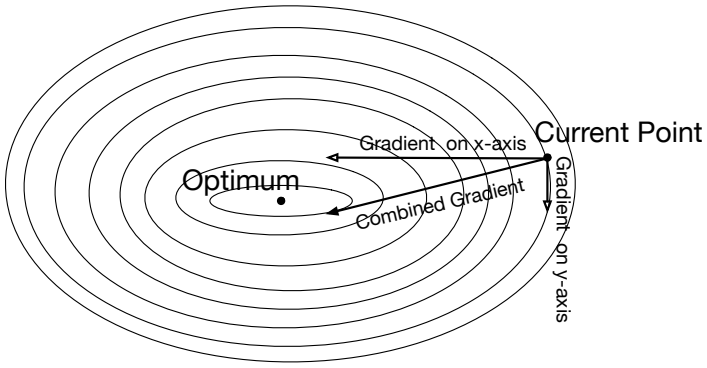


Figure 15.5: Gradient descent training: We compute the gradient with regard to every dimension. In this case the gradient with respect to the y-axis is smaller than the gradient with respect to the x-axis, so we move more to the left than down.

15.1.5 Back-Propagation Training

Learning for neural network means to optimize the weight values so that the network predicts the correct output for a set of training examples. We repeatedly feed the input from training examples into the network, compare the output of the network with the correct output from the training example, and update the weights. Typically, several passes over the training data are carried out.

The most common training method for neural networks is called **back-propagation**, since it first updates the weights to the output layer, and propagates back error information to earlier layers. Whenever a training example is processed, then for each node in the network, an error term is computed which is the basis for updating the values for incoming weights.

The formulae used to compute update values for weights follows principles of **gradient descent** training. The error for a specific node is understood as a function of the incoming weights. To reduce the error given this multi-dimensional function, we compute the gradient of the error term with respect to the weights, and move against the gradient to reduce the error.

Why is moving alongside the gradient a good idea? Consider that we optimize multiple dimensions at the same time. If you are looking for the lowest point in an area (maybe you are looking for water in a desert), and the ground falls off steep in front of you, and also slightly to the right, then you would go mainly forward — and only slightly to the right. In other words, you follow the gradient. See Figure 15.5 for an illustration.

In the following two sections, we will derive the formulae for updating weights. If you are less interested in the *why* and more in the *how*, you can skip these sections and continue reading when we summarize the update formulae on page 9.

Weights to the output nodes

Let us first review and extend our notation. At an output node y_i, we first compute a linear combination of weight and hidden node values.

$$s_i = \sum_j w_{i \leftarrow j} h_j \tag{15.6}$$

This sum s_i is passed through an activation function such as sigmoid to compute the output value y.

$$y_i = \text{sigmoid}(s_i) \tag{15.7}$$

We compare the computed output values y_i against the target output values t_i from the training example. There are various ways to compute an error value E from these values. We use the L2 norm.

$$E = \sum_i \frac{1}{2}(t_i - y_i)^2 \tag{15.8}$$

As we stated above, our goal is to compute the gradient of the error E with respect to the weights w_k to find out in which direction (and how strongly) we should move the weight value. We can do this for each weight w_k separately. We first break up the computation of the gradient into three steps.

$$\frac{dE}{dw_{i\leftarrow j}} = \frac{dE}{dy_i}\frac{dy_i}{ds_i}\frac{ds_i}{dw_{i\leftarrow j}} \tag{15.9}$$

Since we defined the error E in terms of the output values y_i, we can compute the first component as follows.

$$\frac{dE}{dy_i} = \frac{d}{dy_i}\ \frac{1}{2}(t_i - y_i)^2 = -(t_i - y_i) \tag{15.10}$$

The derivative of the output value y_i with respect to s_i (the linear combination of weight and hidden node values) depends on the activation function. In the case of sigmoid, we have:

$$\frac{dy_i}{ds_i} = \frac{d\,\text{sigmoid}(s_i)}{ds_i} = \text{sigmoid}(s_i)(1 - \text{sigmoid}(s_i)) = y_i(1 - y_i) \tag{15.11}$$

To keep our treatment below as general as possible and not commit to the sigmoid as an activation function, we will use the shorthand y_i' for $\frac{dy_i}{ds_i}$ below. Note that for any given training example and any given differentiable activation function, this value can always be computed.

Finally, we compute the derivative of s_i with respect to the weight $w_{i\leftarrow j}$, which turns out to be quite simply the value to the hidden node h_j.

$$\frac{ds}{dw_{i\leftarrow j}} = \frac{d}{dw_{i\leftarrow j}}\sum_j w_{i\leftarrow j}h_j = h_j \tag{15.12}$$

Putting it all together, we have

$$\begin{aligned}\frac{dE}{dw_{i\leftarrow j}} &= \frac{dE}{dy_i}\frac{dy_i}{ds_i}\frac{ds}{dw_{i\leftarrow j}} \\ &= -(t_i - y_i) \quad y_i' \quad h_j\end{aligned} \tag{15.13}$$

Adding in a learning rate μ gives us the following update formula for weight $w_{i\leftarrow j}$.

$$\Delta w_{i\leftarrow j} = \mu(t_i - y_i)\ y_i'\ h_j$$

It is useful to introduce the concept of an **error term** δ_i. Note that this term is associated with a node, while the weight updates concern weights. The error term has to be computed only once for the node, and it can be then used for each of the incoming weights.

$$\delta_i = (t_i - y_i)\ y_i' \tag{15.14}$$

This reduces the update formula to:

$$\Delta w_{i\leftarrow j} = \mu\ \delta_i\ h_j \tag{15.15}$$

Weights to the hidden nodes

The computation of the gradient and hence the update formula for hidden nodes is quite analogous. As before, we first define the linear combination z_j (previously s_i) of input values x_k (previously hidden values h_j) weighted by weights $u_{j\leftarrow k}$ (previously weights $w_{i\leftarrow j}$).

$$z_j = \sum_k u_{j\leftarrow k} x_k \tag{15.16}$$

This leads to the computation of the value of the hidden node h_j.

$$h_j = \text{sigmoid}(z_j) \tag{15.17}$$

Following the principles of gradient descent, we need to compute the derivative of the error E with respect to the weights $u_{j\leftarrow k}$. We decompose this derivative as before.

$$\frac{dE}{du_{j\leftarrow k}} = \frac{dE}{dh_j}\frac{dh_j}{dz_j}\frac{dz_j}{du_{j\leftarrow k}} \tag{15.18}$$

However, the computation of $\frac{dE}{dh_j}$ is more complex than in the case of output nodes, since the error is defined in terms of output values, not values for hidden nodes. The idea behind back-propagation is to track how the error caused by the hidden node contributed to the error in the next layer. Applying the chain rule gives us:

$$\frac{dE}{dh_j} = \sum_i \frac{dE}{dy_i}\frac{dy_i}{ds_i}\frac{ds_i}{dh_j} \tag{15.19}$$

We already encountered the first two terms $\frac{dE}{dy_i}$ (Equation 15.10) and $\frac{dy_i}{ds_i}$ (Equation 15.11) previously. To recap:

$$\begin{aligned}
\frac{dE}{dy_i}\ \frac{dy_i}{ds_i} &= \frac{d}{dy_i}\ \sum_{i'}\frac{1}{2}(t_i - y_{i'})^2 \quad y_i' \\
&= \frac{d}{dy_i}\frac{1}{2}(t_i - y_i)^2 \quad y_i' \\
&= -(t_i - y_i) \quad y_i' \\
&= \delta_i
\end{aligned} \tag{15.20}$$

The third term in Equation 15.19 is computed straightforward.

$$\frac{ds_i}{dh_j} = \frac{d}{dh_j}\sum_i w_{i\leftarrow j} h_j = w_{i\leftarrow j} \tag{15.21}$$

Putting Equation 15.20 and Equation 15.21 together, Equation 15.19 can be solved as:

$$\frac{dE}{dh_j} = \sum_i \delta_i w_{i\leftarrow j} \tag{15.22}$$

This gives rise to a quite intuitive interpretation. The error that matters at the hidden node h_j depends on the error terms δ_i in the subsequent nodes y_i, weighted by $w_{i\leftarrow j}$, i.e., the impact the hidden node h_j has on the output node y_i.

Let us tie up the remaining loose ends. The missing pieces from Equation 15.18 are the second term

$$\frac{dh_j}{dz_j} = \frac{d\,\text{sigmoid}(z_j)}{dz_j} = \text{sigmoid}(z_j)(1-\text{sigmoid}(z_j)) = h_j(1-h_j) = h'_j \tag{15.23}$$

and third term

$$\frac{dz_j}{du_{j\leftarrow k}} = \frac{d}{du_{j\leftarrow k}} \sum_k u_{j\leftarrow k} x_k = x_k \tag{15.24}$$

Putting Equation 15.22, Equation 15.23, and Equation 15.24 together gives us the gradient

$$\begin{aligned} \frac{dE}{du_{j\leftarrow k}} &= \frac{dE}{dh_j}\frac{dh_j}{dz_j}\frac{dz_j}{du_{j\leftarrow k}} \\ &= \sum_i (\delta_i w_{i\leftarrow j}) \;\; h'_j \;\; x_k \end{aligned} \tag{15.25}$$

If we define an error term δ_j for hidden nodes analogous to output nodes

$$\delta_j = \sum_i (\delta_i w_{i\leftarrow j}) \;\; h'_j \tag{15.26}$$

then we have an analogous update formula

$$\Delta u_{j\leftarrow k} = \mu \; \delta_j \; x_k \tag{15.27}$$

Summary

We train neural networks by processing training examples, one at a time, and update weights each time. Weight updates are computed based on error terms δ_i associated with each non-input node in the network.

For output nodes, the error term δ_i is computed from the actual output of the node y_i for our current network, and the target output for the node t_i.

$$\delta_i = (t_i - y_i)\; y'_i \tag{15.28}$$

For hidden nodes, the error term δ_j is computed via back-propagating the error term δ_i from subsequent nodes connected by weights $w_{i\leftarrow j}$.

$$\delta_j = \sum_i (\delta_i w_{i\leftarrow j}) \;\; h'_j \tag{15.29}$$

Computing y'_i and h'_j requires the derivative of the activation function, to which the weighted sum of incoming values is passed.

Given the error terms, weights $w_{i\leftarrow j}$ (or $u_{j\leftarrow k}$) from each proceeding node h_j (or x_k) are updated, tempered by a learning rate μ.

$$\Delta w_{i\leftarrow j} = \mu \; \delta_i \; h_j \tag{15.30}$$

$$\Delta u_{j\leftarrow k} = \mu \; \delta_j \; x_k \tag{15.31}$$

Once weights are updated, the next training example is processed.

Node	Error term	Weight updates
y_0	$\delta = 0.257 \times 0.191 = 0.049$	$\Delta w_{0\leftarrow 0} = \mu \times 0.049 \times 0.731 = 0.036$
		$\Delta w_{0\leftarrow 1} = \mu \times 0.049 \times 0.119 = 0.006$
		$\Delta w_{0\leftarrow 2} = \mu \times 0.049 \times 1.000 = 0.049$
h_0	$\delta = 0.246 \times 0.197 = 0.048$	$\Delta u_{0\leftarrow 0} = \mu \times 0.048 \times 1.000 = 0.048$
		$\Delta u_{0\leftarrow 1} = \mu \times 0.048 \times 0.000 = 0.000$
		$\Delta u_{0\leftarrow 2} = \mu \times 0.048 \times 1.000 = 0.048$
h_1	$\delta = -0.246 \times 0.105 = -0.026$	$\Delta u_{1\leftarrow 0} = \mu \times -0.026 \times 1.000 = -0.026$
		$\Delta u_{1\leftarrow 1} = \mu \times -0.026 \times 0.000 = 0.000$
		$\Delta u_{1\leftarrow 2} = \mu \times -0.026 \times 1.000 = -0.026$

Table 15.2: Weight updates (with unspecified learning rate μ) for the neural network in Figure 15.4 when the training example (1,0) $\rightarrow$ 1 is presented.

Example

Given the neural network in Figure 15.4, let us see when the training example (1,0) $\rightarrow$ 1 is presented.

First, we need to calculate the error term δ for the output node y_0. During inference (recall Table 15.1 on page 5), we computed the linear combination of weighted hidden node values $s_0 = 1.06$ and the node value $y_0 = 0.74$. The target value is $t_0 = 1$.

$$\delta = (t_0 - y_0)\, y_0' = (1 - 0.74) \times \text{sigmoid}'(1.06) = 0.26 \times 0.191 = 0.05 \tag{15.32}$$

With this number, we can compute weight updates, such as for weight $w_{0\leftarrow 0}$.

$$\Delta w_{0\leftarrow 0} = \mu\, \delta_0\, h_0 = \mu \times 0.05 \times 0.73 = \mu \times 0.04 \tag{15.33}$$

Since the hidden node h_0 leads only to one output node y_0, the calculation of its error term δ is not much more computationally expensive.

$$\delta = \sum_i (\delta_i u_{i\leftarrow 0})\ h_0' = (\delta_0 \times w_{0\leftarrow 0}) \times \text{sigmoid}'(z_0) = 0.05 \times 5 \times 0.20 = 0.05 \tag{15.34}$$

Table 15.2 summarizes the updates for all weights.

15.1.6 Refinements

We conclude our introduction to neural networks with some basic refinements and considerations.

Weight initialization Before training starts, the weights are initialized to random values. The values are choses from a uniform distribution.We prefer for training to get started that initial weights lead to values that are in the transition area for the activation function, and not in the low or high shallow slope where it would take a long time to push towards a change. For instance, for the sigmoid activation function, feeding values in the range of, say, $[-1; 1]$ to the activation function leads to activation values in the range of [0.27;0.73].

For the sigmoid activation function, commonly used formula for weights to the final layer of a network are

$$\big[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\big] \tag{15.35}$$

where n is the size of the previous layer and for hidden layers

$$\big[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\big] \tag{15.36}$$

where n_j is the size of the previous layer, n_j size of next layer.

Momentum term Even if most training examples force a weight value into the same direction, it may still take a while for the small updates to accumulate until it reaches its optimum. A common trick is to use a **momentum term** to speed up training for this case. This momentum term m_t gets updated at each time step t (i.e., for each training example). We combine the previous value of the momentum term m_{t-1} with the current raw weight update value Δw_t and use the resulting momentum term value to update the weights.

$$\begin{aligned} m_t &= 0.9 m_{t-1} + \Delta w_t \\ w_t &= w_{t-1} - \mu\, m_t \end{aligned} \tag{15.37}$$

Adapting learning rate per parameter A common training strategy is to reduce the learning rate μ over time. At the beginning the parameters are far away from optimal values, and have to change a lot, but in later training stages we are concerned with fine tuning, and a large learning rate may cause a parameter to bounce around an optimum.

But different parameters may be at different stages of training, so a different learning rate for each parameter may be helpful. One such method, called **Adagrad**, records the gradients that were computed for each parameter and accumulates their square values over time.

The Adagrad update formula based on the gradients of the error E with respect to the weight w at time t, i.e., $g_t = \frac{dE}{dw}$ divides the global learning rate μ by this accumulated sum.

$$\Delta w_t = \frac{\mu}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}}\, g_t \tag{15.38}$$

Intutively, big changes in the parameter value (corresponding to big gradients g_t), lead to a reduction of the learning rate of the weight parameter.

There are various other adaptation schemes, this is an active area of research. For instance, the second order gradient gives some useful information about the rate of change. However, it is hard to compute, so other shortcuts are taken.

Mini batches Each training example yields a set of weight updates Δw_i. We may first process all the training examples and only afterwards apply all the updates. But neural networks have the advantage that they immediately can learn from each training example. A training method that updates the model with each training example is called **online learning**. The online learning variant of gradient descent training is called **stochastic gradient descent**.

Online learning generally takes fewer passes over the training set (called **epochs**) for convergence. However, if the training method constantly changes the weights, it is hard to parallelize training. So, instead, we may want to process the training data in batches, accumulate the weight updates, and then apply them collectively. These smaller sets of training examples are called **mini batches** to distinguish this approach from **batch training** where the entire training set is considered one batch.

There are other variants to organize the processing of the training set, typically motivated by restrictions of parallel processing. If we process the training data in mini batches, then we can parallelize the computation of weight update values Δw, but have to synchronize their summation and application to the weights. If we want to distribute training over a number of machines, it is computationally more convenient to break up the training data in equally sized parts, perform online learning for each of the parts (optionally using smaller mini batches), and then average the weights. Surprisingly, breaking up training this way, often leads to better results than straightforward linear processing.

Vector and matrix operations We can express the calculatons needed for handling neural networks as vector and matrix operations.

- Forward computation: $\vec{s} = W\ \vec{h}$
- Activation function: $\vec{y} = \text{sigmoid}(\vec{h})$
- Error term: $\vec{\delta} = (\vec{t} - \vec{y})\ \text{sigmoid'}(\vec{s})$
- Propagation of error term: $\vec{\delta}_i = W\ \vec{\delta}_{i+1} \cdot \text{sigmoid'}(\vec{s})$
- Weight updates: $\Delta W = \mu\ \vec{\delta}\ \vec{h}^T$

Executing these operations in computationally expensive. If our layers have, say, 200 nodes, then the matrix operation $W\vec{h}$ requires $200 \times 200 = 40,000$ multiplications. Such matrix operations are common also in another highly used area of computer science: graphics processing. When rendering images on the screen, the geometric properties of 3-dimensional objects have to be processed to generate the color values of the 2-dimensional image on the screen. Since there is high demand for fast graphics processing, for instance for the use in realistic looking computer games, specialized hardware has become commonplace: **graphics processing units (GPUs)**.

These processors have a massive number of cores (for example, the NVIDIA Tesla K20c GPU provides 2496 thread processors) but a rather lightweight instruction set. GPUs provide instructions that are applied to many data points at once, which is exactly what is needed out the vector space computations listed above. Programming for GPUs is supported by various libraries, such as CUDA for C, and has become an essential part of developing large scale neural network applications.

15.1.7 Deep learning frameworks

In the next sections, we will encounter various network architectures. What all of these share, however, are the need for vector and matrix operations, as well as the computation of derivatives to obtain weight update formulas. It would be quite tedious to write almost identical code to deal with each these variants. Hence, a number of frameworks have emerged to support developing neural network methods for any chosen problem. At the time of writing, the most prominent ones are **Theano**[1] (a Python library that dymanically generates and compiles C code and is build on NumPy) and Torch[2] (a machine learning library and a script language based on the Lua programming language).

Some of the leading novel neural network methods for machine translation have been developed in Theano. These frameworks are less geared towards ready-to-use neural network architectures, but provide efficient implementations of the vector space operations and computation of derivatives, with seamless support of GPUs. Our example from Section 15.1 can be implemented in a few lines of Python code.

You can execute the following commands on the Python command line interface if you first installed Theano (`pip install Theano`) and included it in the execution environment.

```
> import numpy
> import theano
> import theano.tensor as T
```

The mapping of the input layer `x` to the hidden layer `h` is in terms of a weight matrix `W`, a bias vector `b`, and a mapping function which consists of the linear combination `T.dot` and the sigmoid activation function.

```
> x = T.dmatrix()
> W = theano.shared( value=numpy.array( [[3.0,2.0],[4.0,3.0]] ))
> b = theano.shared( value=numpy.array( [-2.0,-4.0] ))
> x = T.dmatrix() > h = T.nnet.sigmoid(T.dot(x,W)+b)
```

A good way to think about these definitions of `x` and `h` is in term of a functional programming language. They symbolically define operations. To actually define a function that can be called, the Theano method `function` is used.

```
> h_function = theano.function([x], h)
> h_function([[1,0]])
array([[ 0.73105858, 0.11920292]])
```

This example call to `h_function` computes the values for the hidden nodes (compare to the numbers in the table in Section 15.1.4).

The mapping from the hidden layer `h` to the output layer `y` is defined in the same fashion.

```
W2 = theano.shared(value=numpy.array([5.0,-5.0] ))
b2 = theano.shared(-2.0, name='b2')
y_pred = T.nnet.sigmoid(T.dot(h,W2)+b2)
```

Again, we can define a callable function to test the full network.

[1] http://deeplearning.net/software/theano/
[2] http://torch.ch/

```
> predict = theano.function([x], y_pred)
> predict([[1,0]])
array([[ 0.7425526]])
```

Model training requires the definition of a cost function (we use the L2 norm). To formulate it, we first need to define the variable for the correct output. The overall cost is computed as average over all training examples.

```
> y = T.dvector()
> l2 = (y-y_pred)**2
> cost = l2.mean()
```

Gradient descent training requires the computation of the derivative of the cost function with respect to the model parameters (i.e., the values in the weight matrices `W` and `W2` and the bias vectors `b` and `b2`. A great benefit of using Theano is that it computes the derivatives by itself. Note that the following is also an example of a function with multiple inputs and multiple outputs.

```
> gW, gb, gW2, gb2 = T.grad(cost, [W,b,W2,b2])
```

We have now all we need to define training. The function updates the model parameters and returns the current predictions and cost. It uses a learning rate of 0.1.

```
train = theano.function(inputs=[x,y],outputs=[y_pred,cost], \
updates=((W, W-0.1*gW), (b, b-0.1*gb),(W2, W2-0.1*gW2), (b2, b2-0.1*gb2)))
```

All we need now is some training data.

```
> DATA_X = numpy.array([[0,0],[0,1],[1,0],[1,1]])
> DATA_Y = numpy.array([0,1,1,0])
> predict(DATA_X)
array([ 0.18333462, 0.7425526 , 0.7425526 , 0.33430961])
> train(DATA_X,DATA_Y)
[array([ 0.18333462, 0.7425526 , 0.7425526 , 0.33430961]),
array(0.06948320612438118)]
```

The training function returns the prediction and cost before the updates. If we call the training function again, then the predictions and cost change for the better.

```
> train(DATA_X,DATA_Y)
[array([ 0.18353091, 0.74260499, 0.74321824, 0.33324929]),
array(0.06923193686092949)]
```

Typically, we would loop over the training function until convergence. As discussed above, we may also break up the training data into mini-batches and train on one mini-batch at a time.

15.2 Neural Language Models

A good way to think about neural networks is that they 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. We previously addressed this issue 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?).

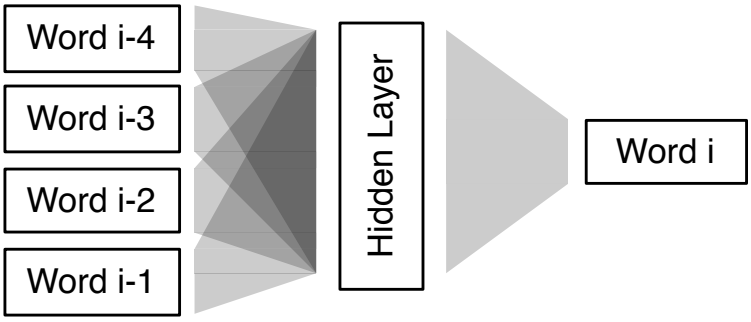


Figure 15.6: Sketch of a neural language model: We predict a word w_i based on its preceding words.

N-gram language models which reduce the probability of a sentence to a word-wise estimation — say, $p(w_i|w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1})$ — 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. Previously, we explored complex discounting and back-off schemes. Now, we turn to neural networks for help.

15.2.1 Feed-Forward Neural Language Models

Figure 15.6 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.

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 124,322 — while in practice these numbers are completely arbitrary.

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. 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 a 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. For now, let us just move on.

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 general continuous space representation for each word, independent of its position in the conditioning context. This representation is commonly referred to as **word embedding**.

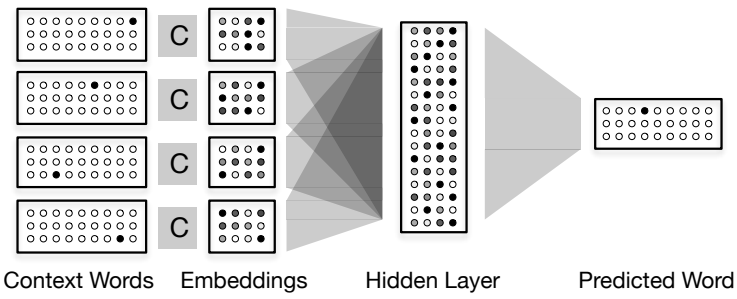


Figure 15.7: Full architecture of a feed-forward neural network language model. Context words $(w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1})$ are represented in a one-hot vector, then projected into continuous space as word embeddings (using the same weight matrix C for all words). The predicted word is computed as a one-hot vector via a hidden layer.

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 closely*
- *child hugged the dog closely*
- *like to watch cat videos*
- *like to watch dog videos*

then the language model would benefit from the knowledge that *dog* and *cat* occur in similar contexts and hence are somewhat interchangeable. Thus, word embeddings enable generalizing between words (clustering) and hence having robust predictions in unseen contexts (back-off).

See Figure 15.7 for a visualization of the architecture the fully fledged feed forward neural network language model. 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.

$$s_i = \sum_j w_{i \leftarrow j} \, h_j \tag{15.39}$$

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.

$$p_i = \text{softmax}(s_i, \vec{s}) = \frac{e^{s_i}}{\sum_j e^{s_j}} \tag{15.40}$$

We train weights 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).

Training such 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. Using the neural language model for machine translation decoding also requires a lot of computation, so we could restrict its use only to re-ranking n-best lists or lattices.

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

- The word embeddings are fixed for the word, 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. 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 15.2.3.

15.2.2 Training

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 neural language models is to increase the likelihood of the training data.

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

$$L(\mathbf{x}, \vec{y}; W) = -\sum_k y_k \log p_k \tag{15.41}$$

Note that only one value y_k is 1, the others are 0. Defining likelihood this way allows us to update all weights, hence also the one that lead to the wrong output words.

For gradient descent training, we have to compute the gradient of the training objective (here: likelihood L) with respect to the weights W. To give the punchline away, for the case of a softmax activation function, the gradient is

$$\frac{dL}{dW} = (\vec{p} - \vec{y})\ \vec{h}^T \tag{15.42}$$

And hence the update with a learning rate μ for each weight w_{ij} is

$$\Delta w_{i \leftarrow j} = \mu\ (p_i - y_i)\ h_j \tag{15.43}$$

If you are not interested in the details how we arrived at this update rule, then skip to the next section. In fact, most papers that report on neural network methods for machine translation do not go into this detail, and may not even explicitly give the weight update formula — just the learning objective and the details of the network (layer sizes, activation functions). Because all that is needed to obtain the update rules given this information is high school calculus. Some software libraries for neural networks even have built-in functionality to compute the gradients, which can become very complex.

Still reading? This is a good chance to brush up on computing derivatives and gives you a more thorough grasp on how neural networks work. So, let's get started.

We expand the computation of the gradient for likelihood L with respect to weights W.

$$\frac{dL}{dW} = \frac{dL}{d\vec{s}}\ \frac{d\vec{s}}{dW} \tag{15.44}$$

The second term $\frac{ds}{dW}$ is easy to compute for each weight $w_{i \leftarrow j}$.

$$\frac{ds_i}{dw_{i \leftarrow j}} = \frac{d}{dw_{i \leftarrow j}} \sum_j w_{i \leftarrow j}\ h_j = h_j \tag{15.45}$$

The first term $\frac{dL}{d\vec{s}}$ takes a bit more work.

$$\begin{aligned}
\frac{dL}{ds_i} &= \frac{d}{ds_i} - \sum_k y_k \log p_k && \textit{(definition of } L\textit{)} \\
&= -\sum_k y_k\ \frac{d}{ds_i} \log p_k && \textit{(}y_k \textit{ is the true value)} \\
&= -\sum_k y_k\ \frac{1}{p_k} \frac{d}{ds_i} p_k && \textit{(chain rule } f(g(x))' = f'(g(x))\ g'(x); log'x = \frac{1}{x}\textit{)} \\
&= -\sum_k \frac{y_k}{p_k}\ \frac{dp_k}{ds_i}
\end{aligned} \tag{15.46}$$

Computing $\frac{dp_j}{ds_i}$ differs for $i = k$ and $i \neq k$. Let us first consider the case $i = k$.

$$\begin{aligned}
\frac{dp_i}{ds_i} &= \frac{d}{ds_i}\frac{e^{s_i}}{\sum_j e^{s_j}} && \textit{(p_i is softmax)} \\
&= \left(\frac{d}{ds_i}e^{s_i}\right)\frac{1}{\sum_j e^{s_j}} + e^{s_i}\left(\frac{d}{ds_i}\frac{1}{\sum_j e^{s_j}}\right) && \textit{(product rule: $(fg)' = f'g + fg'$)} \\
&= \frac{e^{s_i}}{\sum_j e^{s_j}} + e^{s_i}\left(-\frac{\frac{d}{ds_i}\sum_j e^{s_j}}{(\sum_j e^{s_j})^2}\right) && \textit{(reciprocal rule: $(\frac{1}{f})' = -\frac{f'}{f^2}$)} \\
&= \frac{e^{s_i}}{\sum_j e^{s_j}} - \frac{(e^{s_i})^2}{(\sum_j e^{s_j})^2} \\
&= p_i - p_i^2
\end{aligned} \tag{15.47}$$

Now the other case $i \neq k$.

$$\begin{aligned}
\frac{dp_k}{ds_i} &= \frac{d}{ds_i}\frac{e^{s_k}}{\sum_j e^{s_j}} \\
&= e^{s_k}\frac{d}{ds_i}\frac{1}{\sum_j e^{s_j}} \\
&= e^{s_k}\left(-\frac{\frac{d}{ds_i}\sum_j e^{s_j}}{(\sum_j e^{s_j})^2}\right) && \textit{(reciprocal rule: $(\frac{1}{f})' = -\frac{f'}{f^2}$)} \\
&= -\frac{e^{s_k}\, e^{s_i}}{(\sum_j e^{s_j})^2} \\
&= p_k\, p_i
\end{aligned} \tag{15.48}$$

Using Equation 15.47 and 15.48 in Equation 15.46.

$$\begin{aligned}
\frac{dL}{ds_i} &= -\sum_k \frac{y_k}{p_k}\frac{dp_k}{ds_i} \\
&= -\frac{y_i}{p_i}\frac{dp_i}{ds_i} - \sum_{k\neq i}\frac{y_k}{p_k}\frac{dp_k}{ds_i} \\
&= -\frac{y_i}{p_i}(p_i - p_i^2) - \sum_{k\neq i}\frac{y_k}{p_k} - p_k\, p_i && \textit{(plugging in Equation 15.47 and 15.48)} \\
&= -y_i + y_i\, p_i + \sum_{k\neq i} y_k\, p_i \\
&= -y_i + \sum_k y_k\, p_i \\
&= -y_i + p_i \sum_k y_k \\
&= -y_i + p_i && \textit{(only one $y_k = 1$, the others are 0)} \\
&= p_i - y_i
\end{aligned} \tag{15.49}$$

Time to wrap up the solution for Equation 15.44, decomposed into parts Equation 15.45 and Equation 15.46 (continued as Equation 15.49).

$$\frac{dL}{dw_{i \leftarrow j}} = \frac{dL}{ds_i} \frac{ds_i}{dw_{i \leftarrow j}} = (p_i - y_i)\, h_j \tag{15.50}$$

Admittedly, such computations are quite cumbersome. And this is just the gradient for the final layer. The gradients between the hidden layers and the input layer and the hidden have to be carried out similarly. But essentially, it is just calculus. You may use any almost everywhere differentiable activation function or connection structure between layers, calculate the gradient and obtain thus a neural network training method.

15.2.3 Noise Contrastive Estimation

We discussed earlier the problem that computing probabilities with such a model is very expensive due to the need to normalize the 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 $Z(x) \simeq 1$.

$$L(\mathbf{x}, \vec{y}; W) = -\sum_k y_k \log p_k - \alpha \log^2 Z(x) \tag{15.51}$$

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. It requires less computation, 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 a predicted to be a correct training example is

$$p(\text{correct}|\mathbf{x}, \vec{y}) = \frac{p_m(\vec{y}|\mathbf{x}; W)}{p_m(\vec{y}|\mathbf{x}; W) + p_n(\vec{y}|\mathbf{x})} \tag{15.52}$$

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 $p(\text{correct}|\mathbf{x}, \vec{y})$ for noise examples $(\mathbf{x}; \vec{y}) \in U_n$. Using log-likelihood, we define the objective function as

$$L = \frac{1}{2|U_t|} \sum_{(\mathbf{x};\vec{y}) \in U_t} \log p(\text{correct}|\mathbf{x}, \vec{y}) \;\; + \frac{1}{2|U_n|} \sum_{(\mathbf{x};\vec{y}) \in U_n} \log\,(1 - p(\text{correct}|\mathbf{x}, \vec{y})) \tag{15.53}$$

Given the definition of the training objective L, it is back to calculus to compute the gradient $\frac{dL}{dW}$ and the update formula. We skip this lengthy exposition here.

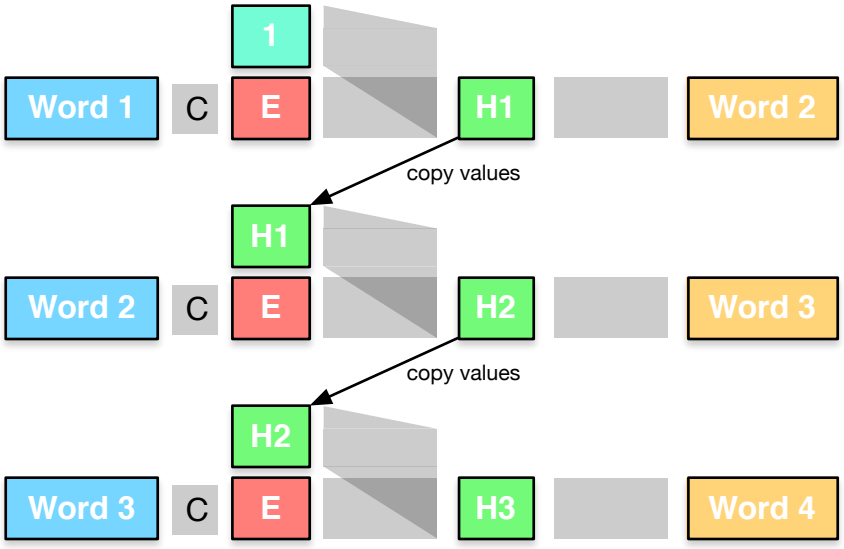


Figure 15.8: Recurrent neural language models: After predicting Word 2 in the context of following Word 1, we re-use this hidden layer (alongside the correct Word 2) to predict Word 3. Again, the hidden layer of this prediction is re-used for the prediction of Word 4.

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 we have encountered before in Chapter . MIRA (margin infused relaxation algorithm) and PRO (pairwise ranked optimization) use the follow the same principle.

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.

15.2.4 Recurrent Neural Language Models

The neural language models that we described so far are able to use longer contexts than traditional statistical back-off models, since they have 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 of any context position. Hence, it is possible to condition on much larger contexts than traditional statistical models. In fact, such 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 15.8 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.

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 roughly, say, 100 real values 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, analogous to 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?

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 15.9) unfolds the recurrent neural network over a fixed number of steps, by going back over, say, 5 word predictions. Architecturally, such a recurrent network unfolded over a fixed length of history, is just a more complex feed-forward neural network with multiple outputs. Note that, despite limiting the unfolding to 5 time steps, the network is still able to learn dependencies over longer distances.

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 15.1.6). First, we process a larger number of training examples (say, 10-20, or the entire sentence), and then update the weights.

15.2.5 Long Short-Term Memory Models

Recurrent neural networks allow modeling of arbitrary 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 allow that.

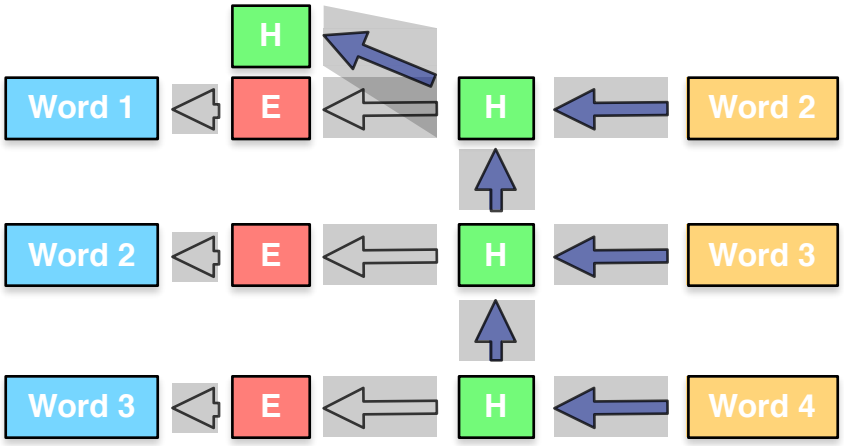


Figure 15.9: Back-propagation through time: By unfolding the recurrent neural network over a fixed number of prediction steps (here: 3), we can derive update formulas based on the training objective of predicting all output words and back-propagation of the error via gradient descent.

- 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 about the **vanishing gradient** — the fact that the impact of recent information at any step drowns out long distance impact.

The rather confusingly named **long short-term memory (LSTM)** neural network architecture addresses these issues. Their design is quite elaborate, although they are not very complicated 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 numbered value.

Furthermore, the read/write/reset operations in a LSTM cell are regulated with a real numbered parameter, which are called gates (see Figure 15.10).

- 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.

$$\begin{aligned} \text{memory}^t &= \text{gate}_{\text{input}} \times \text{input}^t + \text{gate}_{\text{forget}} \times \text{memory}^{t-1} \\ \text{output}^t &= \text{gate}_{\text{output}} \times \text{memory}^t \end{aligned} \tag{15.54}$$

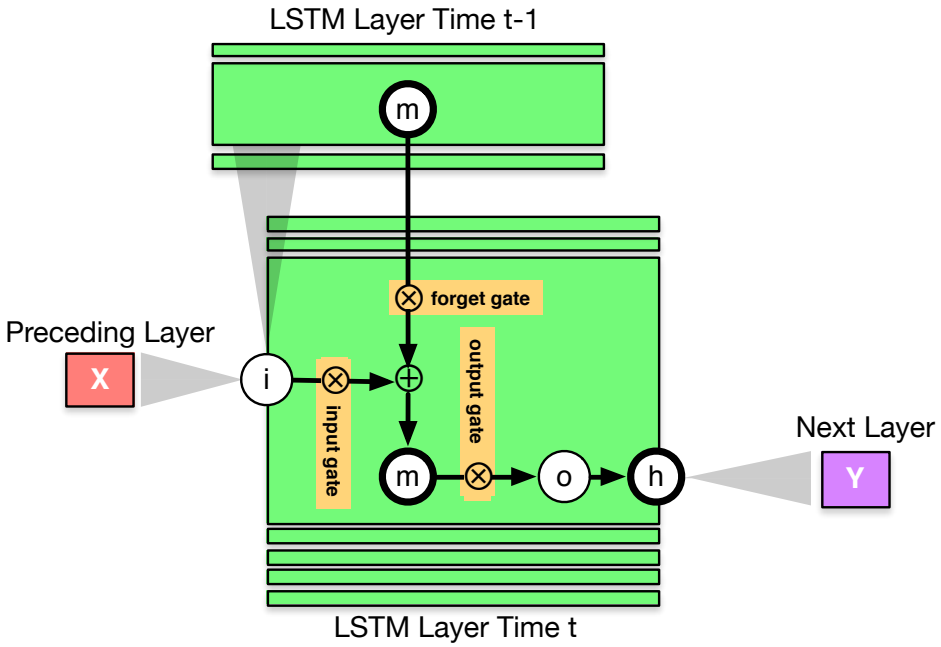


Figure 15.10: A cell in a LSTM neural network. As recurrent neural networks, it receives input from the preceeding layer (x) and the hidden layer values from the previous time step $t-1$. The memory state m is updated from the input state i and the previous time's value of the memory state m^{t-1}. Various gates channel information flow in the cell towards the output value o.

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

$$h^t = f(\text{output}^t) \tag{15.55}$$

The input to a LSTM cell is computed in the same way as the input to a recurrent neural network node. Given the node values for the prior layer $\vec{x}^t = (x_1^t, ..., x_X^t)$ and the values for the hidden layer from the previous time step $\vec{h}^{t-1} = (h_1^{t-1}, ..., h_H^{t-1})$, the input value is the typical combination of matrix multiplication with weights w^x and w^h and an activation function g.

$$\text{input}^t = g\left(\sum_{i=1}^{X} w_i^x x_i^t + \sum_{i=1}^{H} w_i^h h_i^{t-1}\right) \tag{15.56}$$

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 $\vec{x}^t$, the hidden layer $\vec{h}^{t-1}$ at the previous time step, and the memory states at the previous time step $\vec{\text{memory}}^{t-1}$, followed by an activation function h.

$$\text{gate}_a = h\left(\sum_{i=1}^{X} w_i^{xa} x_i^t + \sum_{i=1}^{H} w_i^{ha} h_i^{t-1} + \sum_{i=1}^{H} w_i^{ma} \text{memory}_i^{t-1}\right) \quad (15.57)$$

LSTM are trained the same way as recurrent neural networks, using back-propagation through time. 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.

15.3 Neural Translation Models

Language modeling already challenged simple feedforward neural network architectures, due to the need to model an arbitrarily long sequence. But we have seen that more recent neural network architectures such as recurrent neural networks and long short-term memory neural networks have been developed for such tasks.

The translation problem is even more complex than a sequence prediction task, so we may need different architectures again. In this section, we will discuss auto-encoders and convolutional neural networks. But first, let us start with a simple extension of neural language models to also include translation aspects.

15.3.1 Joint Translation and Language Model

To put it very simply, the task of machine translation involves producing text. Language models give us good indication about which words to produce: the ones that likely follow preceding words. But these new words should also have some relation to the input sentence. Specifically, words in the output sentence should be related to specific words in the input sentence — a notion that we called word alignment. Hence, we can extend a neural language model by adding conditioning context on aligned input words.

Figure 15.11 illustrates the joint translation and language model. Given that we will use this model as an additional feature function in an existing phrase-based model, phrase translations provide possible translation choices and the alignment to words in the input sentence. We add to the conditioning context of a typical n-gram language model (the preceding n words) the aligned input words and a window m words to the left and m words to the right of it.

Training uses a word aligned parallel corpus, but it otherwise similar to the training of feed-forward neural language model. Each training instance consists of predicting a word in the output sentence. The word alignment points to the input word and its surrounding window.

As defined, the joint translation and language model can be used efficiently during decoding. Recall our discussion at the end of Section 15.2.1. This neural network architecture has four layers: an input layer with one-hot vectors for words, a word embedding layer, a hidden layer, and a one-hot output layer. We can not only store the lexicon of word embeddings, but also the impact of each word embedding on the hidden layer. On the output side, we use self-normalizing training to only have to score the new output word that is presented to use by the underlying phrase-based translation model. So we are left with summing up a handful of vectors in the hidden layer and mapping them to one output layer node.

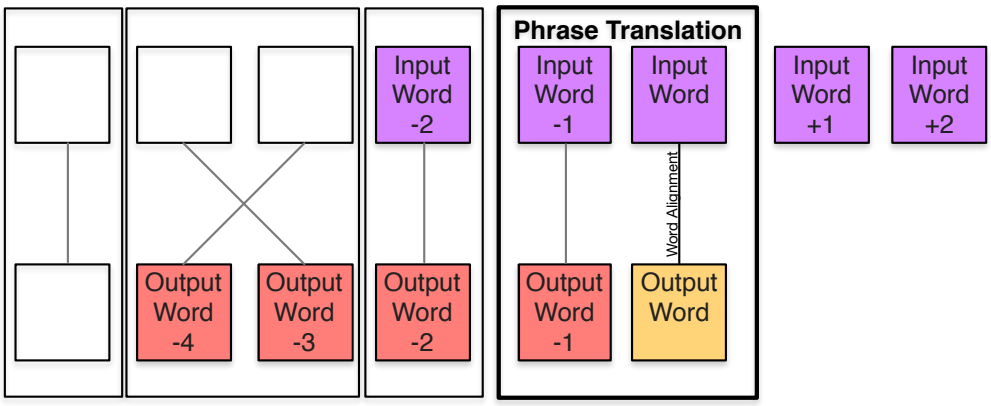


Figure 15.11: Joint translation and language model: In addition to the language model context of the preceding n words (here n=4), word prediction is also conditioned on a m-word (here $m = 2$) window around the aligned input word.

Adding word from the input sentence as additional conditioning context does not make computations much harder. Note that the input words are the same for any possible translation. Their position in the conditioning context may change, but we can pre-compute the impact on the hidden layer for all the input words at any position in the conditioning context window. In fact, input context words cause so little computational overhead that typically more input context words than output context words.

15.3.2 Word and Phrase Translation

The word embedding learned as a by-product of neural language model training are vector representations with the nice property that semantically similar words have similar representations. As an aside, word embeddings may not only be trained in the context of building a neural language model. Instead, we could build a model that predicts a word from its context. For instance, we use (w_{n-2},w_{n-1},w_{n+1},w_{n+2}) as input to predict w_n.

One way to visualize this is to project the high-dimensional vector representation, i.e., the word embedding, down to two dimensions by principle components analysis and then plot the words. See Figure 15.12 for an example, where *horse* and *cow* are closer to each other than *horse* and *cat*. This also holds for the Spanish translations of the words. *Cabello* (Spanish for *horse*) and *vaca* (Spanish for *cow*) are closer to each other than *cabello* and *gato* (Spanish for *cat*).

The graph suggests a intriguing possibility: If we have word embeddings for source and target language (learned from monolingual corpora), can we learn a mapping between the word embeddings to obtain word translations? The graph even suggests that this mapping may as simple as a linear projection (i.e., a matrix multiplication).

This approach allows us to find translations for words that do not occur in the parallel corpus, but may also be used as a feature function in a traditional statistical machine translation model to have more robust estimates for word translations of rare words.

However, translation is not as simple as word substitution. There are many cases where words (especially function words) are added and dropped, multiple words mapped into one or vice versa, and non-compositional idiomatic phrases have translations that do not have a

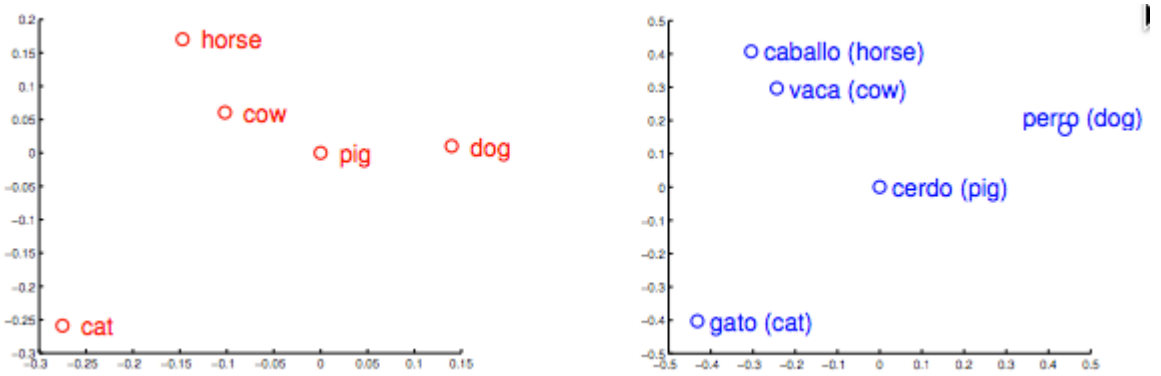


Figure 15.12: Word embeddings for words in English and Spanish (projected down to two dimensions). Similar words have similar representations and appear close to each other in the graph. Words and their translations have similar configurations (taken from Mokolov (2013)).

plausible explanation at the word level. All these are reasons why current statistical machine translation models map multi-word units (so-called phrases) in their atomic translation rules.

If word embeddings are a great idea, how about phrase embeddings?

A starting point to think about this that we need a neural network architecture that maps word embeddings into phrase embeddings. Phrases may be of any length, so we cannot simply have an architecture with a fixed number of input words. One idea is to recursively map two embeddings (word or phrase) into one. So, first we map two neighboring words into a 2-word phrase representation, and then proceed recursively by mapping neighboring phrase or word embeddings to cover the whole phrase.

There are several ways to handle recursion. One way is to find the optimal tree structure. For instance, for a given 3-word phrase (w_1, w_2, w_3), first map w_1 and w_2 into ϕ_{12} and then map ϕ_{12} and w_3 into the full phrase embedding ϕ_{123}. Another way is to compute all possible compositions at every level of the tree, and proceed merging them. For our 3-word phrase this would mean computing both ϕ_{12} and ϕ_{23}, and then using them to compute the phrase embedding ϕ_{123}. The first variant is called **recursive neural network**, and the second variant is called **convolutional neural network**.

How do we know that we have a good phrase embedding?

A characteristic of a good representation is that we can reconstruct the original phrase. This is the idea behind **auto-encoders**. See Figure 15.13 for an illustration of our architecture: First we map the words into a phrase embedding using a convolutional neural network, then we reverse this process to predict the original words again. Note that we have one weight matrix to map two embeddings into one, and another weight matrix to map one embedding into two. To train such a model, all we need is a set of phrases.

Finally, we add a mapping between the phrase embeddings of two different languages. This gives us two learning objectives: One the one hand, we need to satisfy the auto-encoder and obtain phrase embeddings that can reproduce the phrase. On the other hand, the English and foreign word embedding have to easily map to each other. Hence, the model is trained by alternating between the two objective functions and adjusting weights accordingly.

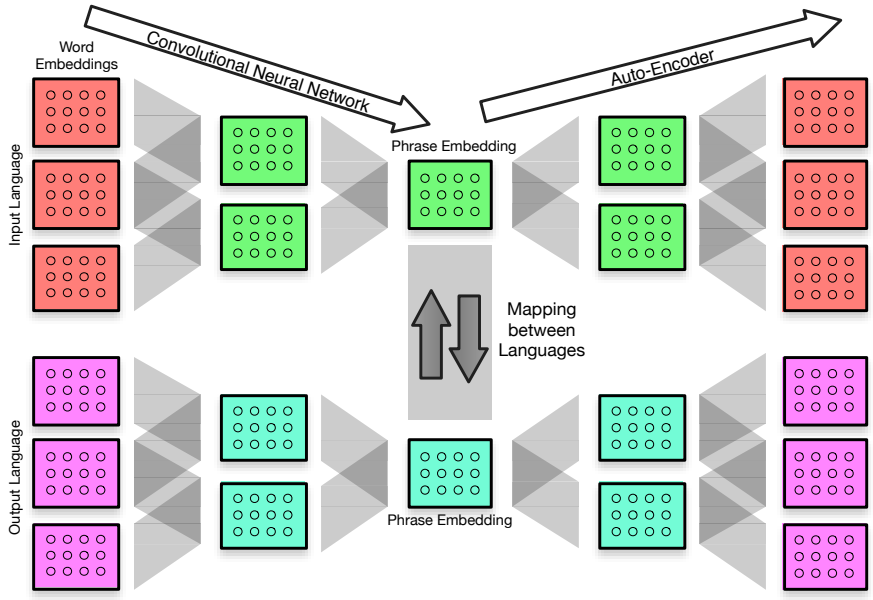


Figure 15.13: Phrase translation model based on convolutional neural networks and auto-encoders. Starting with externally obtained word embeddings, embeddings are combined pairwise using a neural network layer in a neural network architecture called convolutional neural network. The weights for the convolutional neural network are trained with an auto-encoder: the phrase embedding has be expanded again to predict the original word sequence. Phrase embeddings of input and output language are mapped to learn phrase translations.

.

This phrase translation model based on phrase embeddings has to be trained on good phrase pairs. These may be ones with conditional translation probability in either way, or ones used during forced decoding of the parallel corpus.

The resulting model can be used to score each phrase pair of the translation table, thus providing another feature score. Or, we can use it to filter out bad phrase pairs.

At this point you may wonder, why stop at phrase embedding? Why not learn full sentence embeddings and map the entire sentence into the other language? We will come back to this idea in Section 15.5.

15.4 Use of Neural Networks in Other Components

At its heart, neural networks are just another machine learning technique that predicts outputs from inputs. Their strength are more robustness to noisy input and implicit handling of back-off to condition only on some of the inputs. But basically anywhere in our machine translation system where we have conditional probability distributions or log-linear models, we can replace them with a neural network. In this section, we will survey some of these cases, and how special problems with each of them are addressed.

15.4.1 Reordering Model

Traditional statistical machine translation models use two reordering model: one very simple one that just adds up all the jump distances during translation and penalizes the sum with a fixed weight. The other, uses the identity of the phrases (or words at the phrase boundaries) to predict coarse phrase orientation categories such as monotone, swap, or discontinuous (possibly broken down into left/right and short/long jumps).

At the maximum, a lexicalized reordering model would predict orientation between two phrase pairs $(\bar{f}, \bar{e})$ and $(\bar{f}_{-1}, \bar{e}_{-1})$ as:

$$p(\text{orientation}|\bar{f}, \bar{e}, \bar{f}_{-1}, \bar{e}_{-1}) \tag{15.58}$$

With conditional probabilities as modeling technique, we actually do not use such a formulation. Note that if would have actually seen the previous phrase $\bar{f}_{-1}$ and the current $\bar{f}$ next to each other, we may have a larger phrase pair combining both of them, especially if we allow for hierarchical phrases. So, generally, this conditioning context is just too complex. We would have to resort to back-off methods to deal with unseen contexts, which would happen in many if not most cases.

But it is straightforward to formulate this model as a feed-forward neural network. It requires first to compute phrase embeddings, as we have seen in the previous chapter. Then, we use the phrase embeddings as an input layer and the orientation in the output layer.

Since a neural reordering model that is conditioned on current and previous phrase pairs does not impose more restrictions on hypothesis combination than traditional lexicalized reordering, we can integrate it into the decoder. Moreover, we can think of clever pre-computation schemes, since each translation option may occur either as current or previous phrase pair, and we can precompute their impact on the hidden layer of the network.

15.4.2 Operation Sequence Model

The operation sequence model is a reframing of the phrase based translation model as a sequence operations which consist of minimal phrase translations, word insertions and deletions, as well as various jump operations. In the way it conditions translation on both input and output word context, it is not all that different from the joint translation and language model that we discussed in Section 15.3.1. In fact, experiments have shown that the benefits of operation sequence model and joint translation and language model cancel each other out.

We discussed modelling a sequence at length in the context of language model, where we went beyond feed-forward neural networks into recurrent neural networks and long short-term memory networks.

The main challenge that the operation sequence model poses is the representation of the operations, especially the minimal translation units. Besides resorting to convolutional neural networks, there are also simpler idea. Recall that the starting point for word representations was a one-hot vector. If we treat phrase pairs as a bag of words, i.e., ignoring word order within a phrase, we can represent a phrase as a n-hot vector for a n-word phrase. Essentially, this implies adding up of word embeddings.

15.4.3 Dependency Language Model

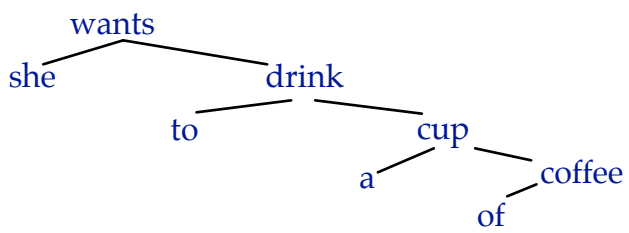
A promise of the syntax-based translation model is that its provide richer annotation of the input and output sentence. With neural networks, we may be able to better take advantage of that annotation.

Take for example the idea of a syntactic language model. Instead of assigning a score to a sentence based on the sequence of n-grams from left to right, we would like to first predict the main verb of the sentence, the subject, object and adjuncts based on the verb, the adjectives based on the nouns, etc. If we have a dependency tree, we have the structure that defines these relationships. The top of the tree is the main verb, and arrows point to dependents recursively.

There are various ways to structure a probabilistic model for such a tree, as long as we pay attention to what we condition to and what we predict — for instance do not predict something that we already condition on. One way to structure the process is top-down, left to right. This means that we predict a words based on its direct ancestors (say, up to 2, i.e., the parent and grand-parent), and its left siblings (say, up to 2).

$$p(\text{word}|\text{parent}, \text{grand-parent}, \text{left-most-sibling}, \text{2nd-left-most-sibling}) \tag{15.59}$$

Consider the following dependency tree for the sentence *She wants to drink a cup of coffee.*



To predict the word *coffee*, we would condition it on its parent *cup*, and grand-parent *drink*, as well at its left sibling *a*. To put it formally:

$$p(\textit{coffee}|\textit{cup}, \textit{drink}, a, \epsilon) \tag{15.60}$$

The problem with traditional statistical models for this distribution is that it is not obvious how to handle back-off. In n-gram models, it is somewhat clear that we want to discard the most distant history first (even if there are skip n-gram models to the contrary). But in the case of the heterogenous conditioning on siblings and parents, what should be discard first? The grand-parent or the oldest sibling?

However, it is straightforward to define this model in a feed-forward neural network. As usual, we first map words to shared embeddings. One specific problem we have here is that we will frequently have empty slots. One common technique to address empty slots is to use the average over all word embeddings in this case.

15.4.4 Feature Combination

Machine translation models are broken up into several model components, as the ones we just discussed above. These are combined in a log-linear model where each component score is given a weight. Since there are complex interactions between these features, the non-linearity of neural networks may be better suited to combine these features.

The input to the neural network are the real-valued feature functions, i.e., the language model score, the forward phrase translation model score, and so one. Using a feed-forward neural network with a hidden layer, we combine these to predict translation quality, as measured by the BLEU score or other machine translation metrics.

Such a non-linear combination cannot be used within the decoder, since the decoding process builds up hypothesis incrementally and relies on the fact that new scores can be simply added to the old scores. So, the neural feature combination can only be used in re-ranking.

15.5 Integrated Neural Machine Translation

In the previous section, we discussed how components of a traditional statistical machine translation system can be replaced with neural network models. In this final section, we take a step further: we do away with the legacy system and start over from scratch to build machine translation system entirely as a neural network.

15.5.1 Encoder-Decoder Approach

The hidden layers of neural networks can be seen as representations of the input distilled to essential information. Think about word embeddings as the meaning representation of words — with certain semantic properties such as words with similar meaning having similar representation. Already hinted upon at the end of Section 15.3, we do not have to stop at words or phrases. Neural network architectures such as recurrent neural networks and convolutional neural networks allow us to encode full sentences.

Take recurrent neural networks for instance. We can turn sentence translation into a sequence prediction problem by lining up the input sentence $e_1, ..., e_n$ and the foreign sentence $f_1, ..., f_m$ as a single sequence to be predicted by a recurrent neural network.

Training data consist of sentence pairs — each a sequence $e_1, ..., e_n, f_1, ..., f_m$ — on which we can train the recurrent neural network, just as we previously trained neural language models. The hidden state after consuming e_n as input encodes the context of the full input sentence. Or to put in other words: it represents the meaning of the input sentence. So, at test time, we can consume the input sentence $e_1, ..., e_n$ and then predict output words one by one until we reach the end of the output sentence.

The length of the output sentence may be decided by the prediction of a end-of-sentence token or by a secondary model that predicts the length of the sentence based on the length and content of the input sentence. A suggested refinement is to line up the output sentence in reverse order, i.e., predicting the sequence $e_n, ..., e_1, f_1, ..., f_m$ which clearly makes it more likely to predict the beginning of the sentence correctly but possibly at the cost of not getting the end right. To overcome any sequential bias, we may use the convolutional neural networks that we en encountered in Section 15.3.2. This neural network architecture parse the sentence bottom-up by combining neighboring constituents to reach a root node that encodes the sentence content.

Such models have shown some promise, both when used to rescore n-best lists of translations from traditional statistical machine translation models and as stand-alone models. But they have also the tendency to get badly off track: generating output sequence that have little if anything to do with the input.

15.5.2 Adding an Alignment Model

To guide the generation of each output word, an explicit alignment to input words may be helpful. We learned from the higher IBM Models and the HMM model for word-based statistical machine translation, that a simple but effective way to condition the alignment of each output word on the alignment of the previous alignment word.

When using the a recurrent neural network to generate output words, then the hidden state of that network contains not only alignment information of previous words, but also the lexical context of the output words. We would like to inform alignment decisions by considering both this history and and a representation of the foreign word. Formally, the alignment $a(i)$ of the output word at position i to the input word f_j at position j depends on the input word and the previous state of the recurrent neural network at that time s_{i-1}.

$$p(a(i) = j | f_j, s_{i-1}) \tag{15.61}$$

There are now several ways how we can use this alignment model in a recurrent neural network that produces the output word sequence. We could identify the most likely aligned input word and then condition on a representation of that word. Or, we could preserve the ambiguity of the alignment and compute a weighted sum of the representations of the aligned words, based on the alignment probability of each.

$$c_i = \sum p(a(i) = j) \times f_j \tag{15.62}$$

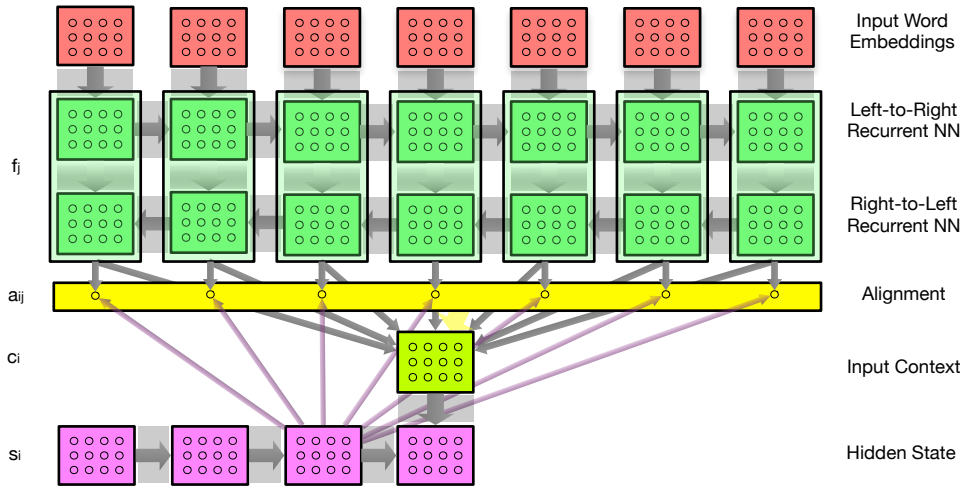


Figure 15.14: The input words are represented by states of two recurrent neural networks — one running from left to right, the other running from right to left. Given this input word representation and the previous output word, we compute a alignment score for each input word. The input words are added up, weighted by their alignment probability to the input context. The recurrent neural network operating on the output side conditions on this input context. Not shown: output words are generated from the hidden state, also conditioned on input context and previous output word.
.

The hidden state of the output recurrent neural network s_i is informed by the previous hidden state s_{i-1} and this encoding of the inout context c_i. From this we can compute the output word e_i.

$$\begin{aligned} s_i &= f(c_i, s_{i-1}, e_{i-1}) \\ e_i &= g(c_i, s_i, e_{i-1}) \end{aligned} \tag{15.63}$$

Let us now finally turn to the representation of the input words. The advantage of recurrent neural networks is that their hidden state does not only encode a single word, but the history up to that word. So, it will be useful to keep a recurrent neural network on the input words and use the hidden states as word representations. Since the left-to-right recurrent neural network encodes each word with its left context, we may also use a right-to-left recurrent neural network to encode each word alongside its right context. So, the representation of input words f_j concatenates the vectors from the left-to-right recurrent neural network $\overrightarrow{f_j}$ and the $\overleftarrow{f_j}$.

See Figure 15.14 how all these pieces fit together.

15.5.3 Adding a Language Model

The models that we have discussed so far integrate translation and language models and train them at the same time. While it is quite compelling to optimize both of them jointly, we typically

have much more monolingual data than parallel data and hence could train the language model on much more data.

So, we first train the large language model as a recurrent neural network on all available data, including the target side of the parallel corpus. Then, we add the model to the neural translation model. Since both neural translation model and neural language model predict output words, the natural point to connect the two models is joining them at that node in the network by joining their conditioning contexts.

We expand Equation 15.5.2 to add the hidden state of the neural language model s_i^{LM} to the hidden state of the neural translation model s_i^{TM}, the source context c_i and the previous English word e_{i-1}.

$$e_i = g(c_i, s_i^{\text{TM}}, s_i^{\text{LM}}, e_{i-1}) \tag{15.64}$$

When training the combined model, we leave the parameters of the large neural language model unchanged, and update only the parameters of the translation model and the combination layer. The concern is that otherwise the output side of the parallel corpus would overwrite the memory of the large monolingual corpus. In other words, the model would overfit to the training data and be less general.

One final question remains: How much weight should be given to the translation model and how much weight should be given to the language model? The above equation considers them equally in all instances. But there may be output words for which the translation model is more relevant (e.g., the translation of content words with distinct meaning) and output words where the language model is more relevant (e.g., the introduction of relevant function words for fluency).

The balance of the translation model and the language model can be achieved with the type of gated units that we encountered in our discussion of the long short-term memory neural network architecture (Section 15.2.5). Such a gated unit may be predicted solely from the language model state s_i^{LM} and is then use as a factor that is multiplied with that language model state before it is used in the prediction of Equation 15.64.

15.5.4 Beam Search

Translating with neural translation models, or predicting any kind of sequence with a neural network model, proceeds one step at a time. We predict the next output word, which gives a probability distribution over the words. We then pick the most likely word and move to the next one. If the model is conditioned on the output word (as is the case in the neural translation model: recall Equation 15.5.2), then we plug in the previous predicted output word.

Such a methodology suggests that we perform 1-best greedy search, which incurs of the so-called garden-path problem that sometimes the best sequence consists first of less probable words which are redeemed by subsequent words which make them the right choices in the context of the full output. In the case of machine translation consider the case of having to produce idiomatic phrases that are non-compositional. The first words of these phrases may be really odd word choices by themselves (e.g., *raining cats and dogs*). Only when the full phrase is formed, then their value is redeemed.

We are faced with the same problem in traditional statistical machine translation models — arguable even more so there since we rely on sparser contexts. In decoding algorithms for these

models, we keep a list of the n-best candidate hypothesis, expand them and keep the n-best expanded hypotheses. We can do the same for neural translation models.

When predicting the first word of the output sentence, we keep a list of the top n most likely choices. Then, we use each of them in the conditioning context for the next word in turn, producing n 2-word sequences for each, for a total of n^2 hypotheses. We prune this list down to the top n 2-word sequences, expand all of them, and so on.

In traditional statistical machine translation we were able to combine hypotheses if they share the same conditioning context for future feature functions. This not possible anymore for recurrent neural networks since we condition on the entire output word sequence from the beginning.

15.5.5 Outlook

Armed with the experience from statistical machine translation, pure neural translation models have reached the stage of IBM Model 2 or the HMM alignment model. They build on the notion of word translation and word alignment, but do not strictly enforce input sentence coverage, incorporate the concept of phrasal translation, and the explicit notion of language as recursive guided by syntactic constraints.

At the time of this writing, it is not clear, if the future of neural machine translation lies in the replacing statistical estimation techniques with neural network layers for all components of traditional phrase-based and syntax-based models, the incorporation of the main insights from these models into the structure of a pure neural network (such as encoding words as morphological feature vectors or use of explicit syntactic annotation such as noun phrases or subjects), or by fully relying on the machine learning capabilities of neural networks by adding more hidden layers, using different architectures and learning algorithms.