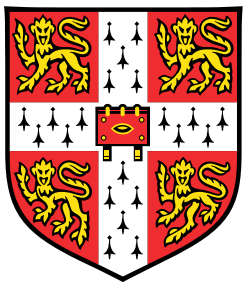
Neural Hidden Markov Models for Word Alignment



Elena Rastorgueva

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Engineering

Corpus Christi College June 2019

Abstract

This project looked into three well-known statistical models for word alignment: the IBM Model 1, IBM Model 2 and HMM, and developed neural forms of these models. It was motivated by the 'Unsupervised Hidden Markov Models' paper by Tran et al. [20] which applied neural HMMs to part of speech tagging.

The neural models developed were trained using a variety of different training methods, most notably Direct Marginal Likelihood (DML) optimisation, which it is generally not possible to apply to the symbolic forms of the alignment models. DML optimisation was applied to all neural models investigated.

The other training techniques applied were: Expectation Maximisation (EM) to HMM, and Viterbi training and supervised learning to IBM Model 1.

This project focused on the English-Spanish language pair. The SciELO corpus, which contains a range of biological and health texts in Spanish and English, was used.

As the IBM Model 1 uses only lexical translation probabilities to form alignments, the neural IBM Model 1 consisted of a neural network for which the input is the embedding of the Spanish source word, and the output is a probability distribution over words in the target (English) vocabulary.

By summing and multiplying the relevant elements of the neural network output, it is possible to calculate the (unnormalised) marginal likelihood of each sentence. This likelihood can be automatically differentiated by a machine learning package (PyTorch was used in this project), and the neural network model parameters will be updated to those that will increase the marginal likelihood of the training data and lead the model to predict more accurate alignments.

The neural IBM Model 1 developed had an Alignment Error Rate (AER) of around 40%, varying with the exact neural architecture used. Alignments produced by GIZA++, a toolkit which can implement symbolic forms of IBM Models 1-5 and HMM, were used as a baseline to calculate the AER of the models developed for this project.

The neural IBM Model 1 was also trained using Viterbi training and supervised learning by optimising the joint probability of the data and alignment variables.

The neural IBM Model 2 consisted of two neural networks: one which modelled the lexical translation probability and took the same form as the neural network used for the neural IBM Model 1, and a second neural network which represented the alignment probability distribution. Positional encodings were used as inputs to this second neural network. The marginal likelihood

of the neural IBM Model 2 can be calculated by element-wise multiplication of relevant probabilities at the outputs of both networks, followed by summation and multiplication in the same fashion as that which was carried out to calculate the marginal likelihood for the neural IBM Model 1.

The neural IBM Model 2 outperformed the neural IBM Model 1, reaching an AER of 31.76%.

The HMM required considerably more computation to calculate marginal likelihood and produce alignments. A lot of computation also was required to carry out EM training.

The HMM applied to word alignment has lexical translation probabilities as emission probabilities (thus we can reuse the IBM Model 1 neural network again), and alignment probabilities as transition probabilities. These transition probabilities can be modelled by a neural network in a similar fashion to the alignment probability model in IBM Model 2, except for the HMM we are considering jumps from one position to another in the source sentence.

Dynamic programming was used to calculate the forward probabilities. The marginal likelihood was obtained by summing the forward probabilities at the end of the HMM trellis. This can be followed by auto-differentiation.

For EM, both forward and backward probabilities had to be calculated, followed by careful multiplication of forward, backward, emission and transition probabilities to calculate the auxiliary function.

The neural HMM also outperformed the neural IBM Model 1, reaching an AER of 37.35% within 5 epochs of training.

Table of contents

Chapter 1

Introduction

This project was inspired by the work of Tran et al. [20], a paper in which the Hidden Markov Model (HMM) tagging model was extended to neural form and used for part of speech tagging.

HMMs can also be used for word alignment of parallel text: the focus of this project was to develop neural models for word alignment, including the simpler IBM Models 1 and 2 as well as the HMM.

Neural word alignment using IBM Models 1 and 2 has been done before (cf. [6], [10], [24]). Some work on neural HMMs for word alignment has also been done [23]. The use of positional encodings for neural alignment models is more novel.

To illustrate what is meant by word alignment in machine translation, an example pair of sentences and their alignments is shown in Figure 1.1. Given pairs of sentences (which are

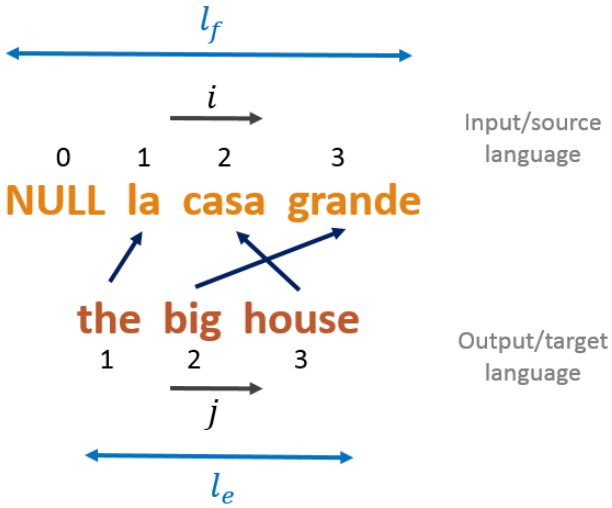


Fig. 1.1 A pair of aligned sentences in English and Spanish, with notation used in this report.

translations of each other) in English and a foreign language (**e** and **f** respectively), alignments, a, are drawn between words with the same meaning. i and j index the foreign and English words respectively, and so the word alignments are mapped by the function $a : j \rightarrow i$. In the given pair

of sentences in Figure 1.1, the alignments are $a : \{1 \to 1, 2 \to 3, 3 \to 2\}$. The number of words in the foreign and English sentence is l_f and l_e respectively. The 'NULL' token is appended to the foreign sentence, to allow for English words which have no corresponding word in the foreign sentence.

In the typical alignment problem set-up, the alignments are not observed, hence this is an unsupervised learning problem.

Numerous models of alignment exist, such as the IBM Models 1-5 [3] and HMM [22]. These each have a generative story to model how words are rearranged and translated in order to go from the source language to the target language. The lexical translation probability of word f being translated to word e, $t(e \mid f)$, is common to all of the models.

In these original formulations, words are treated as symbols, and thus probabilities such as $t(e \mid f)$ are presented in tabular form.

In this project, the probability distributions were represented neural networks. For example, the lexical translation probabilities can be represented using the neural network $t_\theta(e \mid f)$, where the input is the embedding [9] of the source word f, and the output is a probability distribution over words in the target vocabulary. The parametrisation over θ allows us to use gradient-based methods to train the neural alignment model.

Chapter 2

Technical Background

This project investigated the IBM Models 1 and 2 and the Hidden Markov Model. Alignments produced during experiments were compared with alignments produced by GIZA++ [14], a toolkit which can implement symbolic forms of IBM Models 1-5 and HMM.

2.1 General model training

In a general framework, we have training data $\mathbf{x} = \{\mathbf{x}_1, ..., \mathbf{x}_N\}$ which are assumed i.i.d. The training data have unseen, latent variable $\mathbf{z}$. We create a generative model of the data with model parameters, θ, and want to find the parameters which maximise the log likelihood:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\{\mathscr{L}(\theta)\} = \underset{\theta}{\operatorname{argmax}}\{\log p(\mathbf{x};\theta)\} = \underset{\theta}{\operatorname{argmax}}\{\sum_{i=1}^{N} \log p(\mathbf{x}_i;\theta)\} \tag{2.1}$$

The latent variables can also be factorised out to yield:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\{\mathscr{L}(\theta)\} = \underset{\theta}{\operatorname{argmax}}\{\sum_{i=1}^{N} \log \sum_{\mathbf{z}_i} p(\mathbf{x}_i, \mathbf{z}_i;\theta)\} \tag{2.2}$$

2.1.1 Batches and epochs

A distinction worth making explicit is that between batches and epochs. The entire set $\mathbf{x} = \mathbf{x}_1, \ldots, \mathbf{x}_N$ is an epoch, i.e. over an epoch, the entire training set is seen.

Batches are subsets of the entire training set and can be defined using sets of indices $B_b \subset \{1, \ldots, N\}$ such that $\{1, \ldots, N\} = \cup_b B_b$. Then the b^{th} batch is $\mathbf{x}_{B_b} = \{\mathbf{x}_i : i \in B_b\}$ and with batch likelihood

$$\log p(\mathbf{x}_{B_b};\theta) = \sum_{i \in B_b} \log p(\mathbf{x}_i;\theta). \tag{2.3}$$

Batches can vary from epoch to epoch. In this project, the batch assignments are shuffled between each epoch during neural network. This is standard practice as it can speed up convergence [2].

2.1.2 Direct Marginal Likelihood (DML)

For some models, in particular for the neural models to be studied, it is possible to use stochastic gradient descent (SGD) [18] to directly optimise the marginal likelihood of the entire training set. Gradients are accumulated over batches with parameter updates of the form:

$$\begin{aligned}\theta^{t+1} &= \theta^t + \alpha \nabla_\theta \log p(\mathbf{x}_{B_b};\theta)|_{\theta^t} \\ &= \theta^t + \alpha \sum_{i \in B_b} \nabla_\theta \log p(\mathbf{x}_i;\theta)|_{\theta^t}\end{aligned} \tag{2.4}$$

The gradient of each sample $\mathbf{x}_i$ is calculated 'directly' by marginalising the latent variables: $\nabla_\theta \log p(\mathbf{x}_i;\theta)|_{\theta^t} = \nabla_\theta \log \sum_{\mathbf{z}_i} p(\mathbf{x}_i,\mathbf{z}_i;\theta)|_{\theta^t}$. This can be done easily using machine learning frameworks such as PyTorch, which has auto-differentiation capabilities [16].

2.1.3 Expectation Maximisation

In cases when it is not possible use DML, Expectation Maximisation (EM) [4] is often used instead. EM generates a sequence of parameters θ^p such that $\log p(\mathbf{x};\theta^{p+1}) > \log p(\mathbf{x};\theta^p)$, until convergence.

The EM auxiliary function is

$$\begin{aligned}Q(\theta,\theta^p) &= E_{\mathbf{z}|\mathbf{x};\theta^p} \log p(\mathbf{x},\mathbf{z}|\theta) \\ &= \sum_{\mathbf{z}} p(\mathbf{z}|\mathbf{x};\theta^p) \log p(\mathbf{x},\mathbf{z}|\theta) \\ &= \sum_i \underbrace{\sum_{\mathbf{z}} p(\mathbf{z}|\mathbf{x}_i;\theta^p) \log p(\mathbf{x}_i,\mathbf{z}|\theta)}_{Q_i(\theta,\theta^p)} = \sum_i Q_i(\theta,\theta^p)\end{aligned} \tag{2.5}$$

and θ^{p+1} is chosen via the M-Step as

$$\theta^{p+1} = \operatorname*{argmax}_\theta Q(\theta,\theta^p). \tag{2.6}$$

Alternatively, there is the Generalised Expectation Maximisation algorithm (GEM) [7], which uses the fact that if $\theta^{p+1} : Q(\theta^{p+1},\theta^p) > Q(\theta^p,\theta^p)$ then $\log p(\mathbf{x};\theta^{p+1}) > \log p(\mathbf{x};\theta^p)$. The steps for **GEM with the M-step completed via gradient ascent** are as follows:

1. Fix θ^p from the previous epoch.
2. Note that $\nabla_\theta Q(\theta,\theta^p) = \sum_i \nabla_\theta Q_i(\theta,\theta^p)$.
3. Perform SGD to increase $Q(\theta,\theta^p)$ until θ' is found so that $Q(\theta',\theta^p) > Q(\theta^p,\theta^p)$.

(a) SGD can be performed over batches, as in DML:
An iterative estimate of θ' can be produced from θ'' as:

$$\theta' = \theta'' + \alpha \sum_{i \in B_b} \nabla_\theta Q_i(\theta, \theta^p)|_{\theta''} \tag{2.7}$$

(b) Stop when $Q(\theta', \theta^p) > Q(\theta^p, \theta^p)$ which should be verified over all $\mathbf{x}$, not just the current batch.

4. $\theta^{p+1} \leftarrow \theta'$.

Many forms of approximations are possible, such as carrying out multiple iterations per batch or forgoing verifying 3.b over the entire set. One variant which was used in this project is to recalculate the posterior after each batch. This violates the EM guarantees for improved likelihood, but is computationally simpler.

2.1.4 Viterbi training

Viterbi training, sometimes refered to as 'hard' EM, yields a sequence of parameters θ^p such that $\max_{\mathbf{z}} \log p(\mathbf{x}, \mathbf{z}; \theta^{p+1}) > \max_{\mathbf{z}} \log p(\mathbf{x}, \mathbf{z}; \theta^p)$. The steps of Viterbi training are:

1. For θ^p find $\mathbf{z}^p = \text{argmax}_{\mathbf{z}} \log p(\mathbf{x}, \mathbf{z}; \theta^p)$.
2. Perform iterations of SGD with Supervised Learning with $\mathbf{z}^p$ as $\hat{\mathbf{z}}$.
3. If θ' can be found such that $\log p(\mathbf{x}, \mathbf{z}^p; \theta') > \log p(\mathbf{x}, \mathbf{z}^p; \theta^p)$, then $\theta^{p+1} \leftarrow \theta'$. Like EM, this should be verified for the entire training set, and not just at the batch level.

2.1.5 Supervised Learning

If the latent variable $\hat{\mathbf{z}}_i$ is given for each observed $\mathbf{x_i}$, the objective is to maximise

$$\log p(\mathbf{x}, \hat{\mathbf{z}}; \theta) = \sum_i \log p(\mathbf{x}_i, \hat{\mathbf{z}}_i; \theta)$$

Parameter updates are carried out as in DML, except that it is not necessary to marginalise over $\mathbf{z}_i$, as we know the true value of $\hat{\mathbf{z}}_i$.

$$\begin{aligned} \theta^{t+1} &= \theta^t + \alpha \nabla_\theta \log p(\mathbf{x}_{B_b}, \hat{\mathbf{z}}_{B_b}; \theta)|_{\theta^t} \\ &= \theta^t + \alpha \sum_{i \in B_b} \nabla_\theta \log p(\mathbf{x}_i, \hat{\mathbf{z}}_i; \theta)|_{\theta^t} \end{aligned} \tag{2.8}$$

2.2 From the general formulation to alignment models

The training methods described above can be applied to alignment models, with the following changes in notation:

$$p(\mathbf{x}) \rightarrow p(\mathbf{e} \mid \mathbf{f})$$

$$p(\mathbf{x}, \mathbf{z}) \rightarrow p(\mathbf{e}, a \mid \mathbf{f})$$

$$p(\mathbf{z} \mid \mathbf{x}) \rightarrow p(a \mid \mathbf{e}, \mathbf{f})$$

Here it is assumed that the models are conditioned given the foreign sentence **f**, that the English sentence **e** is the 'observed' variable, and that the alignment a is the latent, or missing, variable.

Numerous models of alignment exist, such as the IBM Models 1-5 [3] and HMM [22]. These each have a generative story to model how words are rearranged and translated in order to go from the source language to the target language, so they each have different expressions for $p(\mathbf{e} \mid \mathbf{f}), p(\mathbf{e}, a \mid \mathbf{f}), p(a \mid \mathbf{e}, \mathbf{f})$, which will be introduced in the following sections.

2.3 Symbolic IBM Model 1

The IBM Model 1 considers only lexical translation probabilities. Thus the probability of observing a sentence **e** with alignments a, given a source sentence **f** is:

$$p(\mathbf{e}, a \mid \mathbf{f}) = \frac{\varepsilon}{(l_f + 1)^{l_e}} \prod_{j=1}^{l_e} t(e_j \mid f_{a(j)}), \tag{2.9}$$

where ε is an arbitrary normalising constant. The equations for the marginal likelihood and the posterior are:

$$p(\mathbf{e} \mid \mathbf{f}) = \sum_a p(\mathbf{e}, a \mid \mathbf{f}) = \frac{\varepsilon}{(l_f + 1)^{l_e}} \prod_{j=1}^{l_e} \sum_{i=0}^{l_f} t(e_j \mid f_i), \tag{2.10}$$

$$p(a \mid \mathbf{e}, \mathbf{f}) = \prod_{j=1}^{l_e} \frac{t(e_j \mid f_{a(j)})}{\sum_{i=0}^{l_f} t(e_j \mid f_i)}. \tag{2.11}$$

Full derivations of these equations are in [3].

The optimal alignment, $\hat{a}$, is given by:

$$\hat{a} = \operatorname*{argmax}_a p(\mathbf{e}, a \mid \mathbf{f}) = \operatorname*{argmax}_a \prod_{j=1}^{l_e} t_\theta(e_j \mid f_{a_j}) \tag{2.12}$$

Because the alignment $a(j)$ from each target word j is independent of the alignments for all other words, we can write the following equation for $\hat{a}(j)$.

$$\hat{a}(j) = \operatorname*{argmax}_i t_\theta(e_j \mid f_i) \tag{2.13}$$

As stated earlier, when training a model, we wish to maximise its marginal likelihood.

It is not possible to do this directly with the symbolic IBM Model 1. However, an iterative estimation procedure based on Expectation Maximisation can be obtained [3].

2.3.1 EM training

The symbolic IBM Model 1 is normally trained using Expectation Maximisation by initialising the model to some random values $t(e_j \mid f_i)$. These are used to calculate the posterior, $p(a|\mathbf{e},\mathbf{f})$, as part of the E step. The M step involves calculating the number of times that each word e aligns with f, weighed by the probability of that alignment occurring. This calculation is written, and simplified, as follows:

$$\begin{aligned} c(e|f;\mathbf{e},\mathbf{f}) &= \sum_a p(a \mid \mathbf{e},\mathbf{f}) \sum_{j=1}^{l_e} \delta(e,e_j)\delta(f,f_{a(j)}) \\ &= \frac{t(e \mid f)}{\sum_{i=0}^{l_f} t(e \mid f_i)} \sum_{j=1}^{l_e} \delta(e,e_j) \sum_{i=0}^{l_f} \delta(f,f_{a(j)}). \end{aligned} \tag{2.14}$$

The counts can then be used to re-calculate the translation probabilities as follows:

$$t(e \mid f;\mathbf{e},\mathbf{f}) = \frac{\sum_{(\mathbf{e},\mathbf{f})} c(e|f;\mathbf{e},\mathbf{f})}{\sum_{\mathbf{e}} \sum_{(\mathbf{e},\mathbf{f})} c(e|f;\mathbf{e},\mathbf{f})}. \tag{2.15}$$

The steps are repeated until convergence in likelihood.

2.3.2 Supervised learning

It is quite interesting to consider the supervised learning case, which could, for example, be used to initialise the model before it continues with some other, unsupervised mode of training. The training data $\hat{a}$ can, for example, be obtained from alignments generated by GIZA++.

Supervised learning for the symbolic IBM Model 1 is simply a case of counting the number of times pairs of words are aligned, and normalising to form a valid probability distribution for $t(e \mid f)$. This only needs to be done once, without iteration.

2.3.3 Viterbi training

Viterbi training is essentially hard EM. Whereas EM training is 'soft' because counts are calculated by weighing by the probability of every possible alignment, in Viterbi training, counts are calculated by considering only most probable alignment. Thus equation (2.14) evolves into the following:

$$c(e|f;\mathbf{e},\mathbf{f}) = \sum_{j=1}^{l_e} \delta(e,e_j)\delta(f,f_{\hat{a}(j)}). \tag{2.16}$$

Like EM, Viterbi training is done iteratively, with estimates for $t(e \mid f)$ and $c(e|f;\mathbf{e},\mathbf{f})$ obtained alternately. Estimates for $t(e \mid f)$ are obtained by equation 2.15.

2.4 Symbolic IBM Model 2

Training and decoding are similar to IBM Model 1, but with an additional alignment probability model which accounts for how alignments tend to be created between word positions in the source and target language. The notation $p_{M2}(i \mid j, l_e, l_f)$ will be used to represent the probability of an English word in position j being aligned with a foreign word in position i, conditioned on the sentence lengths l_e & l_f.

The joint and marginal likelihood expressions are:

$$p(\mathbf{e}, a \mid \mathbf{f}) = \varepsilon \prod_{j=1}^{l_e} t(e_j \mid f_{a(j)})\ p_{M2}(a(j) \mid j, l_e, l_f) \tag{2.17}$$

$$p(\mathbf{e} \mid \mathbf{f}) = \sum_a p(\mathbf{e}, a \mid \mathbf{f}) = \varepsilon \prod_{j=1}^{l_e} \sum_{i=0}^{l_f} t(e_j \mid f_i)\ p_{M2}(i \mid j, l_e, l_f) \tag{2.18}$$

Derivations of these equations are, again, in [3].

The optimal alignment, $\hat{a}$, is given by:

$$\hat{a} = \underset{a}{\operatorname{argmax}}\ p(\mathbf{e}, a \mid \mathbf{f}) = \underset{a}{\operatorname{argmax}} \prod_{j=1}^{l_e} t(e_j \mid f_{a(j)})\ p_{M2}(a(j) \mid j, l_e, l_f) \tag{2.19}$$

This can be decomposed into the following expression:

$$\hat{a}(j) = \underset{i}{\operatorname{argmax}}\ t(e_j \mid f_i)\ p_{M2}(i \mid j, l_e, l_f) \tag{2.20}$$

2.4.1 EM training

EM is normally used to train the symbolic IBM Model 2. The procedure is similar to the symbolic IBM Model 1, with the formula for fractional counts $c(e \mid f;\mathbf{e},\mathbf{f})$ modified to account for alignment distortion, and the introduction of a formula for fractional counts for alignment distortion: $c(i \mid j, l_e, l_f;\mathbf{e},\mathbf{f})$ [3].

The equation for the posterior is now:

$$p(a \mid \mathbf{e},\mathbf{f}) = \frac{p(\mathbf{e}, a \mid \mathbf{f})}{p(\mathbf{e} \mid \mathbf{f})} = \prod_{j=1}^{l_e} \frac{t(e_j \mid f_{a(j)})\ p_{M2}(a(j) \mid j, l_e, l_f)}{\sum_{i=0}^{l_f} t(e_j \mid f_i)\ p_{M2}(i \mid j, l_e, l_f)} \tag{2.21}$$

2.5 Symbolic HMM

Word alignment can be modelled using a Hidden Markov Model [22], where the observed state at 'time' j (or rather, position j), is the English word e_j. The hidden state at 'time' j is the Spanish word which 'emitted' the English word e_j. Using our notation, this Spanish word is denoted $f_{a(j)}$. Possible hidden state sequences are shown in Figure 2.1, with the most probable sequence shown using black arrows.

In this HMM set-up, the emission probability, i.e. the probability that word $f_{a(j)}$ emits e_j, is simply the lexical translation probability $t(e_j \mid f_{a(f)})$, which was used in the IBM Models. The transition probability is the probability that the hidden state at 'time' j is i, given the hidden state at 'time' $j-1$ is i'. Thus these transition probabilities will be denoted as $p_{HMM}(i \mid i')$.

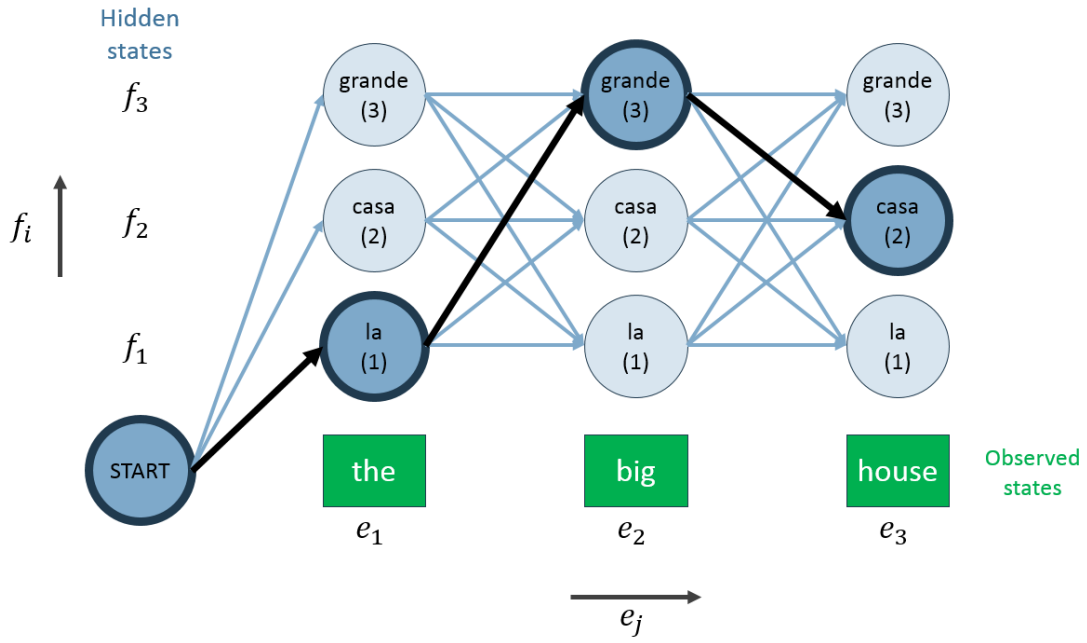


Fig. 2.1 Hidden Markov Model trellis for the observed sequence 'the big house'. The correct hidden state sequence 'START la grande casa' is highlighted.

Using general HMM notation of $\mathbf{x}$ & $\mathbf{z}$ to represent observed and latent variables respectively, the probability of an n-length sequence of observed and latent variables is:

$$p(\mathbf{x},\mathbf{z}) = \prod_{i=1}^{n} p(z_t|z_{t-1}) \prod_{i=1}^{n} p(x_t|z_t) \tag{2.22}$$

Using the notation we have introduced in the context of word alignment, this joint probability is expressed as:

$$p(\mathbf{e},a \mid \mathbf{f}) = \prod_{j=1}^{l_e} t(e_j \mid f_{a(j)}) \prod_{j=1}^{l_e} p_{HMM}(a(j) \mid a(j-1)) \tag{2.23}$$

For HMMs, we will also need to use dynamic programming to calculate forward and backward probabilities $\alpha_i(j)$ and $\beta_i(j)$ for hidden state i at 'time' j. These are expressed as follows:

$$\alpha_i(j) = P(e_1, e_2, ...e_j, f_{a(j)} = i \mid \theta) \tag{2.24}$$

$$\beta_i(j) = P(e_{j+1}, e_{j+2}, ..., e_{l_e} \mid f_a(j) = i, \theta) \tag{2.25}$$

The posterior alignment probability is now:

$$p(a \mid \mathbf{e}, \mathbf{f}) = \frac{p(\mathbf{e}, a \mid \mathbf{f})}{p(\mathbf{e} \mid \mathbf{f})} = \frac{\prod_{j=1}^{l_e} \Big(t(e_j \mid f_{a(f)})\, p_{HMM}(a(j) \mid a(j-1) \Big)}{\sum_{i=0}^{l_f} \alpha_i(l_e)} \tag{2.26}$$

The Viterbi path probability $v_i(j)$, used to keep track of the most probable path through the hidden states. Some detail of how this is implemented (for the neural case) is shown in Figure 3.2.

2.5.1 EM training

The Baum-Welch algorithm, a form of EM, is normally used to train the symbolic HMM [1].

The E step involves the calculation of the state occupancy count γ and expected state transition count ξ as follows (using notation as in [12]):

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\alpha_T(q_F)} \tag{2.27}$$

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\alpha_t(q_F)} \tag{2.28}$$

The M step involves the calculation of emission and transition probabilities as follows [12]:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{t=k}^{N} \xi_t(i,k)} \tag{2.29}$$

$$\hat{b}_j(v_k) = \frac{\sum_{t=1 \, s.t. \, o_t = v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{2.30}$$

2.6 Neural IBM Model 1

A neural network was used to model the lexical translation probabilities, $t_\theta(e_j \mid f_i)$, where θ represents the parameters of the neural network. The input to the neural network is an embedding of the source word, and the output is a probability distribution of target language words. The word embeddings (of size $D_{embed,M1}$) were initialised randomly, and allowed to be updated

during training, a procedure which is acceptable given enough data [9]. Alternatively, pre-trained word embeddings could have been used.

The neural IBM Model 1 pipeline is shown in the diagram in Figure 2.2. For simplicity, the diagram indicates the neural network as containing a single hidden layer of size H_{M1}, although the neural network could have any configuration. A softmax is always applied at the output layer to produce a valid probability distribution.

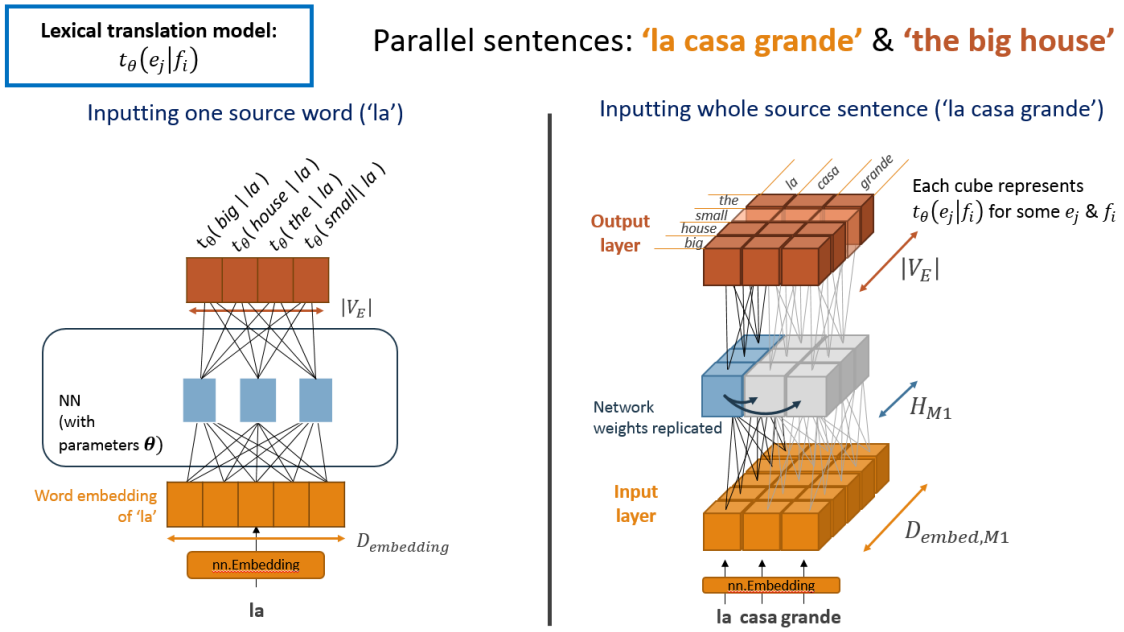


Fig. 2.2 Diagram of lexical translation probability neural network. Left-hand side: generating translation probabilities for a single input word. These can be stacked together and run in parallel to generate translation probabilities for every word in a source sentence, as shown on the right-hand side.

The simple form of equation (2.13) means that decoding (i.e. producing word alignments) is straightforward. To produce alignments, we pass through the target words, and for each target word j, we look up the row of that word in the output layer, and output the foreign word which corresponds to the location of the highest probability value in that row.

2.6.1 DML training

Because the neural model is parametrised by θ, it is possible to maximise the marginal likelihood directly, using the following derivatives:

$$J_{DML}(\theta) = \frac{\partial}{\partial \theta} \log p_\theta(\mathbf{e} \mid \mathbf{f}) \tag{2.31}$$

By inspecting equation (2.10), we can see that $J_{DML}(\theta)$ can be calculated by multiplying and summing over the relevant $t_\theta(e_j \mid f_i)$ terms. These values are then passed through through an auto-differentiation package [16].

Stochastic gradient descent is then used to tune the neural network to settle on the $t_\theta(e_j \mid f_i)$ function which minimises the loss function, and thus maximises the sentence likelihood.

2.6.2 EM training

EM can also be formulated for the neural model. In the general EM case, the aim is to maximise the auxiliary function $\mathbb{E}_{p(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{x},\mathbf{z} \mid \theta)]$ [7] . Thus the gradient used in updates is [20]:

$$J(\theta) = \sum_{\mathbf{z}} p(\mathbf{z} \mid \mathbf{x}) \frac{\partial}{\partial \theta} \log p(\mathbf{x}, z \mid \theta). \tag{2.32}$$

In the case of the alignment problem, this translates to:

$$J_{EM}(\theta) = \sum_{a} p(a \mid \mathbf{e},\mathbf{f}) \frac{\partial}{\partial \theta} \log p_\theta(\mathbf{e}, a \mid \mathbf{f}). \tag{2.33}$$

For the proper EM procedure, care should be taken on when values are updated: $p(a \mid \mathbf{e},\mathbf{f})$ should only be updated after we have seen all of the training data (i.e. after each epoch, not each batch). If it can be shown that the auxiliary function increases over all the data, then it is guaranteed that the likelihood increases as well.

2.6.3 Supervised learning

For the neural case, the aim is to maximise the joint probability $p(\mathbf{e},\hat{a} \mid \mathbf{f})$, leading to the following gradient:

$$J_S(\theta) = \frac{\partial}{\partial \theta} \log p(\mathbf{e},\hat{a} \mid \mathbf{f}) \tag{2.34}$$

For the IBM Model 1, this can be thought of as a classification problem, with the word pairs in the training data forming input and target values. A cross entropy loss function is used, as is typical for classification, which penalises the model if the value of $t(e_j \mid f_{\hat{a}(j)})$ is low.

This forms one of the terms to be summed in the expression for the log-likelihood:

$$\log p(\mathbf{e},\hat{a} \mid \mathbf{f}) = K + \sum_{j=1}^{l_e} \log t(e_j \mid f_{\hat{a}(j)}) \tag{2.35}$$

Thus we can see that if a cross entropy loss function is used [11], the gradient update is equivalent to the expression in equation (2.34).

2.6.4 Viterbi training

In contrast with the iterative updates of equations (2.16) and (2.15) which are used for Viterbi training in the symbolic case, in the neural case, Viterbi training is carried out using the following

gradient:

$$J_V(\theta) = \frac{\partial}{\partial \theta} \log p(\mathbf{e}, \hat{a} \mid \mathbf{f}) \tag{2.36}$$

Comparing equation (2.36) with equation (2.34), we can see that neural Viterbi training is like supervised learning, with the 'training data' being the alignments that the model currently thinks are most probable.

2.7 Neural IBM Model 2

Now we need to use two neural networks: one to represent lexical translation probabilities (i.e. the neural IBM Model 1), and one to represent the alignment probability distribution $p_{M2}(i \mid j, l_e, l_f)$. Whereas the inputs of the first network are word embeddings, the second network uses positional embeddings [21] of size $D_{embed,M2}$. 4 sets of positional embeddings are created to represent each of i, j, l_e and l_f, and then fed into the neural network. This is repeated for each possible combination of i and j, and the outputs are normalised to produce valid $p_{M2,\theta}(i \mid j, l_e, l_f)$ probability distributions.

Sinusoidal positional encodings, as described in [21], were used. The elements of the embedding matrix PE satisfy the following equations:

$$PE_{(pos,k\,even)} = \sin\left(\frac{pos}{10{,}000^{\frac{2k}{d_{model}}}}\right), \tag{2.37}$$

$$PE_{(pos,k\,odd)} = \cos\left(\frac{pos}{10{,}000^{\frac{2k}{d_{model}}}}\right). \tag{2.38}$$

where pos indexes the rows and k indexes the columns in the PE matrix.

The pipeline for processing one sentence is shown in Figure 2.3. Again, we are assuming the simplest case, in which the neural network has one hidden layer, of size H_{M2}.

2.7.1 DML training

As for the IBM Model 1, training the neural IBM Model 2 is possible by DML, and follows a very similar procedure to Model 1. By inspecting equation (2.18), we can see that the likelihood can be calculated by element-wise multiplication of the output probabilities of the alignment distortion model, and of the output probabilities of the lexical translation model (after foreign words which are not in the target sentence are masked). The relevant summation and multiplication is then carried out, prior to auto-differentiation.

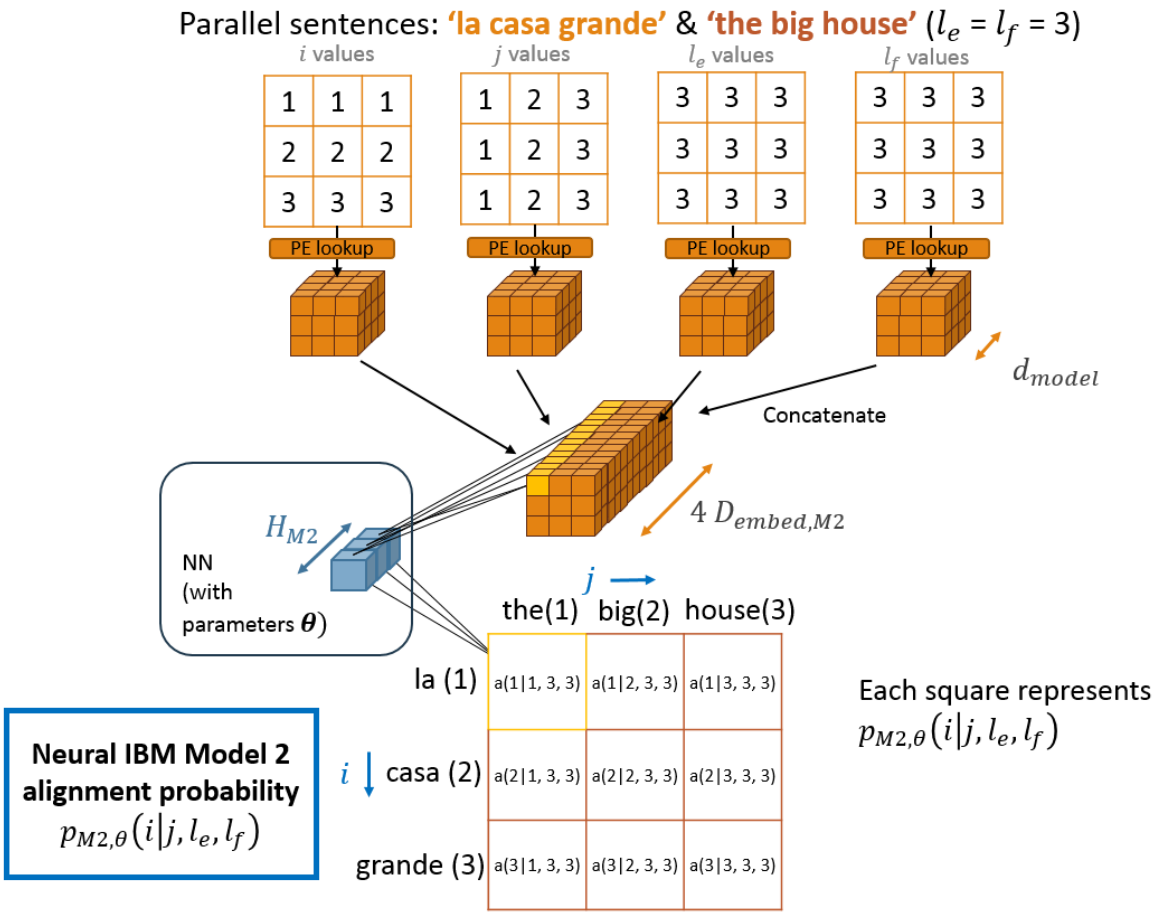


Fig. 2.3 Diagram of alignment distortion neural network. Four 2D matrices containing positional encoding keys become four 3D tensors after passing through the embedding stage. The four tensors are concatenated to produce the input for the neural network. We can see that there are $l_e \times l_f$ (in the diagram, $l_e \times l_f = 9$) encodings of possible alignments. Each encoding has dimension $4d_{model}$. The neural network is applied to each encoding of possible alignments. To save space, the diagram shows the neural network being shown for the possible alignment $(i = 1, j = 1)$, highlighted in yellow.

2.7.2 Supervised learning & Viterbi training

Supervised learning and Viterbi training can be applied to the IBM Model 2 in similar ways as for IBM Model 1. The log likelihood, which is needed for the neural implementation, is now:

$$\log p(\mathbf{e}, \hat{a} \mid \mathbf{f}) = K + \sum_{j=1}^{l_e} \left(\log t(e_j \mid f_{\hat{a}(j)}) + \log p_{M2}(\hat{a}(j) \mid j, l_e, l_f \right) \tag{2.39}$$

This simply requires the summation of relevant terms at the outputs of the lexical translation and alignment distortion neural networks. The parameters are then optimised via gradient descent.

2.8 Neural HMM

In earlier sections, we have already created a neural network which can model the emission probabilities $t_\theta(e_j \mid f_i)$. The transition probabilities can be modelled by a neural network in a similar fashion to the alignment distortion model in IBM Model 2, except this time we are considering jumps from one position to another in the source sentence. We also condition the model on the length of the foreign word sentence. The pipeline for obtaining the transition matrix values, $p_{HMM,\theta}(i|i',l_f)$, is shown in Figure 2.4. Here we denote the size of positional embeddings as $D_{embed,HMM}$, and the size of the single hidden layer as H_{HMM}. The diagram is simplified: the actual implementation also contains a 'start' state to allow us to obtain estimates for the initial distribution. Forward & backward probabilities, and Viterbi path probabilities

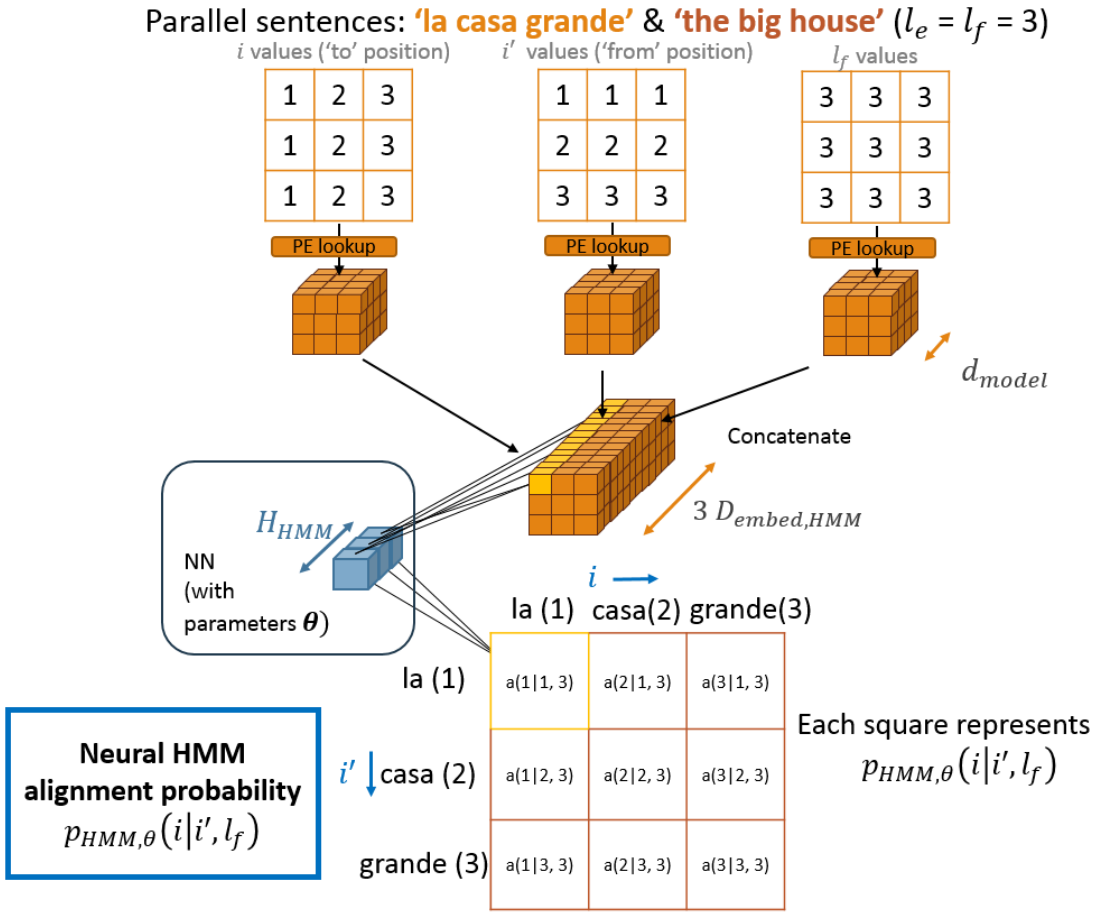


Fig. 2.4 Diagram of transition probability neural network for a neural HMM. The pipeline is essentially identical to the alignment distortion model for IBM Model 2, except conditioning is on the prior alignment variable rather than source position, and there is no conditioning on l_e.

can be calculated by careful manipulation of the outputs of the neural networks. Dynamic programming needs to be used.

2.8.1 DML training

The neural HMM can be trained by DML, as the marginal likelihood can be calculated by summing the forward probabilities at the end of the HMM trellis:

$$p(\mathbf{e} \mid \mathbf{f}) = \sum_{i=0}^{l_f} \alpha_i(l_e) \tag{2.40}$$

2.8.2 EM training

Alternatively, as discussed in [20], a neural Hidden Markov Model can be trained by differentiating the auxiliary function. Using general HMM notation, the derivatives to be calculated can be expressed as follows (this is the extended form of equations in [20]):

$$\begin{aligned}
J(\theta) &= \sum_{\mathbf{z}} p(\mathbf{z} \mid \mathbf{x}) \frac{\partial}{\partial \theta} \log p(\mathbf{x}, \mathbf{z} \mid \theta) \\
&= \sum_{\mathbf{z}} \frac{p(\mathbf{z}, \mathbf{x})}{p(\mathbf{x})} \frac{\partial}{\partial \theta} \Big(\sum_t \log p(x_t \mid z_t, \theta) + \sum_t \log p(z_t \mid z_{t-1}, \theta) \Big) \\
&= \sum_t \sum_{z_t} \frac{p(z_t, \mathbf{x})}{p(\mathbf{x})} \frac{\partial}{\partial \theta} \log p(x_t \mid z_t, \theta) + \sum_t \sum_{z_t} \sum_{z_{t-1}} \frac{p(z_t, z_{t-1}, \mathbf{x})}{p(\mathbf{x})} \frac{\partial}{\partial \theta} \log p(z_t \mid z_{t-1}, \theta) \\
&= \sum_t \sum_{z_t} \frac{\alpha_{z_t}(t) \beta_{z_t}(t)}{p(\mathbf{x})} \frac{\partial}{\partial \theta} \log p(x_t \mid z_t, \theta) \\
&\quad + \sum_t \sum_{z_t} \sum_{z_{t-1}} \frac{a_{z_{t-1}}(t-1) p(z_t \mid z_{t-1}) \beta_{z_t}(t) p(x_t \mid z_t)}{p(\mathbf{x})} \frac{\partial}{\partial \theta} \log p(z_t \mid z_{t-1}, \theta)
\end{aligned} \tag{2.41}$$

As noted previously, in the proper EM procedure, the posterior $p(\mathbf{z} \mid \mathbf{x})$ needs to remain unchanged between batches, and only updated at the end of each epoch, after all of the training data has been seen.

Using word alignment notation, the gradients can be expressed as follows:

$$\begin{aligned}
J(\theta) &= \sum_j \sum_i \frac{\alpha_i(j) \beta_i(j)}{p(\mathbf{e} \mid \mathbf{f})} \frac{\partial}{\partial \theta} \log t_\theta(e_j \mid f_i) \\
&\quad + \sum_j \sum_i \sum_{i'} \frac{a_{i'}(j-1) a(i \mid i') \beta_i(j) t(e_j \mid f_i)}{p(\mathbf{e} \mid \mathbf{f})} \frac{\partial}{\partial \theta} \log p_{HMM,\theta}(i \mid i')
\end{aligned} \tag{2.42}$$

2.8.3 Supervised learning

The HMM can be trained using supervised learning. The joint log likelihood takes the following form:

$$\log p(\mathbf{e}, \hat{a}, \mid \mathbf{f}) = \sum_{j=1}^{l_e} \Big(t_\theta(e_j \mid f_{\hat{a}(j)}) + p_{HMM,\theta}(\hat{a}(j) \mid \hat{a}(j-1)) \Big), \tag{2.43}$$

hence all that is needed is a cross entropy loss functions between the emission & transition probabilities and the 'true' emissions and transitions, which can be determined from the alignment training data.

2.8.4 Viterbi training

Viterbi training takes a similar form to supervised learning, but the training data is replaced with the current most likely alignment predicted by the model.

This can be accomplished by completing Viterbi decoding to find $\hat{a}$, and applying auto-differentiation to the Viterbi path probability at the output node of $\hat{a}$ on the Viterbi trellis

2.9 Measuring quality of alignment

An often referenced measure of word alignment quality is the alignment error rate (AER) [15], which considers sure and possible alignment points. As GIZA made no distinction between the two, a modified version of this measure was created based on GIZA++'s alignments, G, and the alignments, $\hat{A}$, predicted by the models created for this project neural alignments:

$$AER = 1 - \frac{2 \mid G \cap \hat{A} \mid}{\mid G \mid + \mid \hat{A} \mid}. \tag{2.44}$$

$\mid G \mid$, for example, represents the number of alignments produced by GIZA++.

Chapter 3

Design of experiment

3.1 Language data

The parallel corpus used was a subset of the SciELO dataset, a corpus of over 700,000 sentences which contains a range of biological and health texts in Spanish and English.

3.2 GIZA++ alignments

The performance of every model developed was measured against the alignments produced by GIZA++ trained on the entire SciELO corpus. GIZA++ can produce 1:n alignments or symmetrized n:m alignments. For this project, the 1:n English to Spanish alignments were used (in agreement with the alignment setup introduced in Figure 1.1).

For each sentence pair, GIZA++ produced three lines: one line indicating the sentence pair number, sentence lengths and an alignment score; one line stating the English sentence; one line stating each Spanish e_i with the index (or indices) j of the English word to which it was aligned.

Thus the GIZA++ output for the sentence pair 'the big house' and 'la casa grande' would be:

```
Sentence pair (1) source length 3 target length 3 alignment score:  XXX
the big house
NULL ( ) la ( 1 ) casa ( 3 ) grande( 2 )
```

All the information necessary to extract the alignments is contained in the third line (as long as we keep track of the relative orders of sentences). Therefore every third line of the GIZA output file was processed to produce an array containing an array of indices j for each word in position i. The array for the example above would be:

```
[ [ ] , [0], [2], [1]]
```

It is worth noting that indexing used by GIZA++ starts at 1, whereas it was decided to use programming convention and start indexing at 0 for the alignment arrays. The arrays for each sentence were themselves contained within a larger array in Python which contained the GIZA alignments for every relevant sentence.

Different arrays were created for training sets (for supervised training), the 'test set' (which is a subset of the training set, as is customary in word alignment) and the 'held out set' which contains only sentences that were never seen during training.

After sentences that are too long were discarded (this itself is discussed later), the indices of the 'test set' were generated by randomly selecting 1,000 numbers between 1 and 50,000. The indices of the 'held out set' were generated by randomly selecting 1,000 numbers between 500,000 and 600,000, as it was decided that certainly no model would be trained on more that 500,000 sentences. A random seed of value '1' was set before each of these selection operations, allowing this selection to be replicated across different experiments.

3.3 Computing resources

PyTorch was used for the neural network programming [16]. Most of the code was written and run using Google Colaboratory [1], which allowed free use of a Tesla K80 (updated to a Tesla T4 in the latter half of the project), although it had the downside of a runtime limitation of one hour, making long computations difficult. Google Colaboratory, however, was sufficient to carry out the development of the code.

Once the code was completed and ready for full experiments to be run, some of it was executed on the CUED Speech Group cluster. This was particularly useful for experiments that took several hours to run.

3.4 Symbolic Model 1 implementation

The first body of work completed for this project was to write code in Python to perform EM training of the symbolic IBM Model 1. The pseudocode provided in [13] was used as a guide. Versions using probabilities calculated in the natural and logarithmic domain were produced, with no discernible performance difference.

3.5 Neural IBM Model 1 implementation

The next step was to implement the neural IBM Model 1, starting with DML training. Although the first implementation was naively implemented using one-hot encoding of input words, natural domain probabilities (instead of log probabilities) and batch sizes of 1, a more-sophisticated final implementation is described here.

Due to memory constraints (i.e. to ensure the number of output nodes, V_E, was not so large as to cause Out-Of-Memory (OOM) errors), the corpus was limited to the 50,000 most common words, with all rarer words replaced with an 'OOV' token. Byte-pair encoding (BPE) [19] was

[1] Google Colaboratory can be accessed at this web address: https://colab.research.google.com/

briefly discussed as a means of getting round the large vocabulary size, but it was not used due to the difficulty of generating alignments for the IBM Models. As removing infrequent words is a long process, this was done once, and the modified text was saved and used in all further experiments.

The sentence length in all experiments was limited to no more than 50 words in both the English and Spanish sentences for each sentence pair. As required by GIZA++, 'NULL' tokens were added to the start of each Spanish sentence, and 'DUMMY' tokens were placed in any empty sentences. It was arbitrarily chosen that the sentence length limit would be applied after the 'NULL' and 'DUMMY' tokens were applied.

The sentence length limit was applied due to memory constraints aswell, i.e. to ensure the number of input nodes ($l_{f,max}$) was not so large as to cause OOM errors on the GPU. Filtering out long sentences is a quick operation and was done at the start of every experiment by taking the intersect of the indices of each of the English and Spanish sentences which met the criteria. This left more than 600,000 sentences which met the criteria.

3.5.1 Feed forward pipeline

The raw data at the start of each experiment is the sentence-aligned SciELO corpus (with limited vocabulary and sentence length, as described above).

The training data was then selected. For all experiments in this report these were the first 50,000 sentences. The vocabulary used in each language training set was found by creating a Python `set` of all of the words it contained. The list was then sorted (a not essential step, but one which was deemed to be useful for intuition and debugging). A dictionary was then created for each language, with the key being the word string, and the value being the word's location in the vocabulary. This dictionary was used to create a 'lookup array' for each sentence. An example set of variables is (assuming $l_{f,max} = 4$ across the entire dataset, and that our vocabulary V_E contains only 'casa', 'grande', 'la', 'pequeña'):

```
   e_sentences[0] = 'la casa grande'
lookup_array_e[0] = [3, 1, 2, 0]
```

The zero padding (note the `0` above to account for the disparity between l_f and $l_{f,max}$) is necessary to allow us to feed in sentences of different lengths within each batch.

The Pytorch `Dataset` class is used to define how data should be picked out, given a sentence index which is picked by the Pytorch `DataLoader` class (which simplifies dataloading, requiring only parameters such as batch size, and whether shuffling is to be used). The `Dataset` class also allows features to be extracted on the fly, such as determining sentence length by counting the number of non-zero entries in the 'lookup array'[2].

[2]More detail on the use of the classes described can be found at this web address: https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

The first stage in the neural network pipeline is the embedding layer. A PyTorch module which automatically produces random embeddings given an index is used. The embeddings are updated as training goes on.

The hidden layer, activation and log softmax at the output are applied quite trivially. The activation layer was chosen to be the Rectified Linear Unit, the most widely-used activation function [17]. A log softmax was used to switch operation to the log probability space, to improve numerical stability. To remove the risk of weights being initialised sub-optimally, the weights were initialised using Xavier initialisation [8] at the start of all experiments.

One nuance during model evaluation is that when the neural network is applied to the held out test data, we need to be aware of the fact that the held out data contains words outside the model's vocabulary. A small script was written to convert these words to 'OOV' tokens at the start of each experiment.

3.5.2 DML of neural IBM Model 1

The fact that sentences in a batch have different lengths had to be accounted for carefully for each mode of training. This is because zeros in lookup arrays get converted to [0, ..., 0] embeddings, but they have a non-zero output at the final layer (due to bias terms). These outputs are arbitrary, and we want to make sure that we do not include them in our DML calculation. Subject to the constraints of PyTorch, the implementation used for these experiments involved replacing rows and columns for absent English or Spanish words (respectively) with either `torch.tensor([-float(np.inf)])` or `torch.tensor([float(0)])` depending on whether the next operation on that element is `torch.logsumexp` or `torch.sum` respectively. These are the log probability space equivalents of addition or multiplication, respectively, in the probability space.

Although the proper expression for the marginal likelihood (cf. equation (2.10) includes the constant $\frac{\varepsilon}{(l_f+1)^{l_e}}$, this expression was not included during calculations, as it is independent of the model's parameters. Thus when the 'marginal likelihood' is referenced in the context of model training, it is often an unnnormalised marginal likelihood value.

Once calculated, the DML can be auto-differentiated using PyTorch's SGD functionality [16].

3.5.3 Generating and evaluating model alignments

Model alignments were generated sentence-by-sentence. This process was much slower than training due to the use of for-loops to iterate over sentences and words within sentences.. Different `DataLoader` and `DataSet` objects had to be used for these, with shuffling turned off so that the sentences could be matched with their GIZA++ counterparts.

For each sentence, a dictionary was created with a key for each Spanish word. Values were initialised to an empty array. The code then looped through each English word, found the Spanish

word corresponding to the highest lexical translation probability, and appended the index i of the English word to value array of the j-th key.

For the {'NULL la casa grande', 'the big house'} example, the dictionary with the correct alignment would be:

```
0:  [ ], 1:  [0], 2:  [2], 3:[1]
```

The keys are necessary during the alignment generation process, but are not needed subsequently. We can initialise an empty model alignment array, and then pass through each key in numeric order, appending its value to the model alignment array. This array is now in the same format as the array extracted from GIZA++.

To calculate the AER, we loop through each sentence and compare alignment arrays, maintaining a running total of $|\, G \,|, |\, \hat{A} \,|$ and $|\, G \cap \hat{A} \,|$. After all relevant arrays have been compared, equation (2.44) is applied.

3.5.4 Viterbi training of neural IBM Model 1

An additional `wordpairDataset` class was defined (by extending the `Dataset` class) so that specific word pairs could be fed in. The word pairs were generated from model alignments, which themselves were generated by reusing the code mentioned above. The `NLLLoss` (negative log likelihood loss) function provided by PyTorch was used to train the model on the alignments which it considers most probable.

The Viterbi training used here was a slight modification of that described in section 2.1.4, because SGD was applied on a batch-by-batch basis, so it was only guaranteed that the joint likelihood of that batch data would be increased.

3.5.5 Supervised learning of neural IBM Model1

The implementation was very similar to that for Viterbi training. Word pairs were produced from the GIZA++ alignment matrices. As word pairs only needed to be produced once, as opposed to every epoch, supervised training was faster than Viterbi training.

3.6 Neural IBM Model 2 implementation

Implementing the neural IBM Model 2 involved completing the feed forward pipeline for the neural IBM Model 1 output as described above, to produce an output matrix which we shall call `t_pred`. A separate neural network was created to model alignment probabilities $p_{M2,\theta}(i \mid j, l_e, l_f)$, for which the structure is shown in Figure 2.3.

To obtain the positional encodings during training, a matrix `PE` was created with element values determined by equations (2.37) and (2.38).

The matrix that was outputted for each batch of the second neural network is called `p_M2_pred`. To calculate the model 2 DML, the outputs of the two neural networks need to be multiplied correctly.

A derivative of the `t_pred` matrix, `t_pred_useful`, was obtained as shown in Figure 3.1. This is in the correct form to carry out element-wise addition (it is addition and not multiplication, because we are in the log probability space at this stage) between `t_pred_useful` and `p_M2_pred`. The probabilities that result are the probabilities needed to predict alignments (cf. 2.20). The elements are then summed and multiplied in the same way as for IBM Model 1.

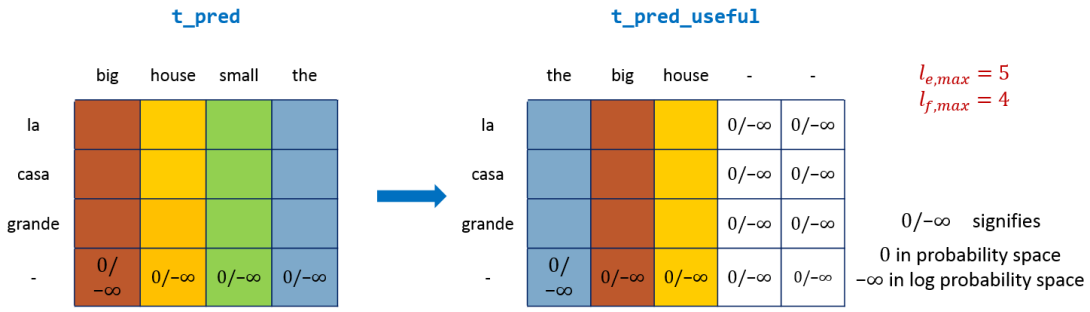


Fig. 3.1 Diagram to show how `t_pred` is transformed into `t_pred_useful` (simplified to a one-sentence example - in actual fact both matrices are 3-dimensional).

3.7 Neural HMM implementation

The feed-forward aspects of the neural HMM were very similar to the neural IBM Model 2. A lot of thought had to be put into the next steps: how to carry out the forward algorithm (to calculate the marginal likelihood), how to carry out Viterbi decoding (to produce alignment predictions), and how to calculate posterior probabilities (in order to carry out Expectation Maximisation). Some of these aspects are included in this section.

3.7.1 DML for neural HMM

This required the forward algorithm to be completed, after which the sum of the forward probabilities at the end of the HMM trellis were used to compute the model's gradients (cf. equation 2.40). A diagram of how this was done is shown in Figure 3.2. Note that the initialisation probability matrix (commonly denoted as π) has been incorporated into the transition matrix, which has the variable name `p_HMM_pred`.

Viterbi decoding then had to be completed in order to produce the model's alignment predictions, so that the AER could be calculated.

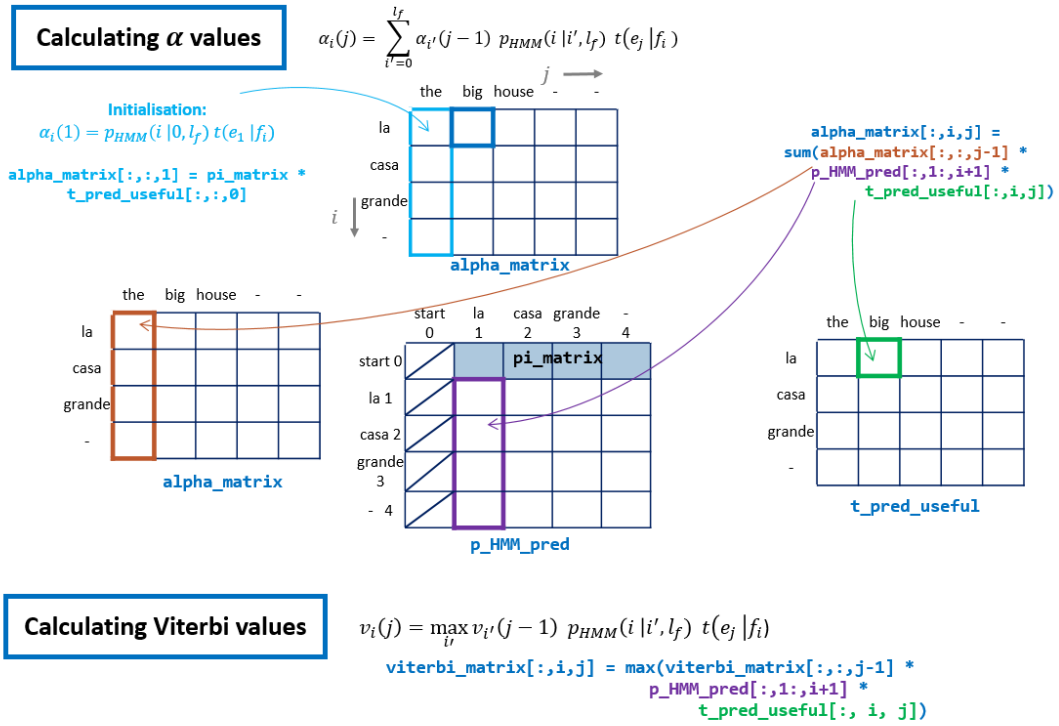


Fig. 3.2 Calculation of α values, which are stored in a $(batchsize \times l_{f,max} \times l_{e,max})$ matrix `alpha_matrix`, drawn here without the first dimension).

3.7.2 EM

Carrying out EM required the calculation of β values, which were calculated in a similar fashion to the α values, but required extra care regarding the variable sentence length (as β had to be initialised to a probability of 1 at the last word in the English sentence. A diagram of how dynamic programming of β values was implemented is shown in Figure 3.3.

The auxiliary function was calculated as in equation (2.42). The most difficult step to visualise is the calculation of $a_{i'}(j-1)a(i \mid i')\beta_i(j)t(e_j \mid f_i)$ before summation over i', i, j is carried out. This was done by creating a 3D matrix as shown in Figure 3.4.

The optimisation technique used was a form of GEM - with each step it was not attempted to maximise the auxiliary function, only to find a set of parameters which increased it.

An attempt was made to accumulate the posteriors over the entire epoch, however, this resulted in numerical instability. A simpler attempt in which the posteriors varied with each batch (as discussed in section 2.1.3) gave a converging result: this is the result shown in this report.

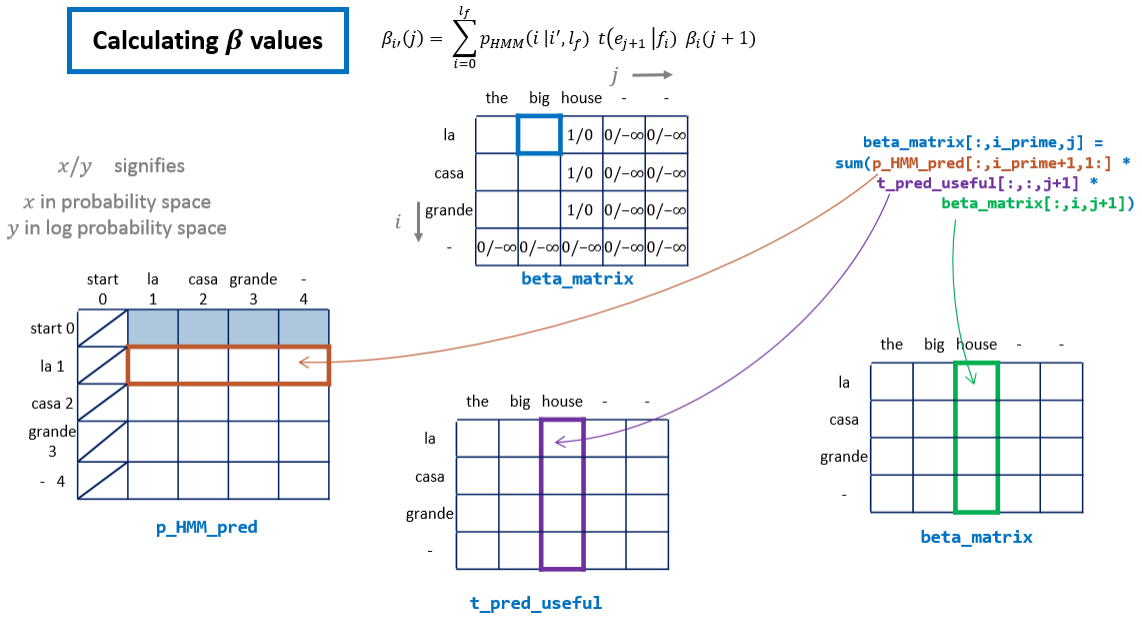


Fig. 3.3 Calculation of β values, which are stored in a $(batchsize \times l_{f,max} \times l_{e,max})$ matrix `alpha_beta`, drawn here without the first dimension).

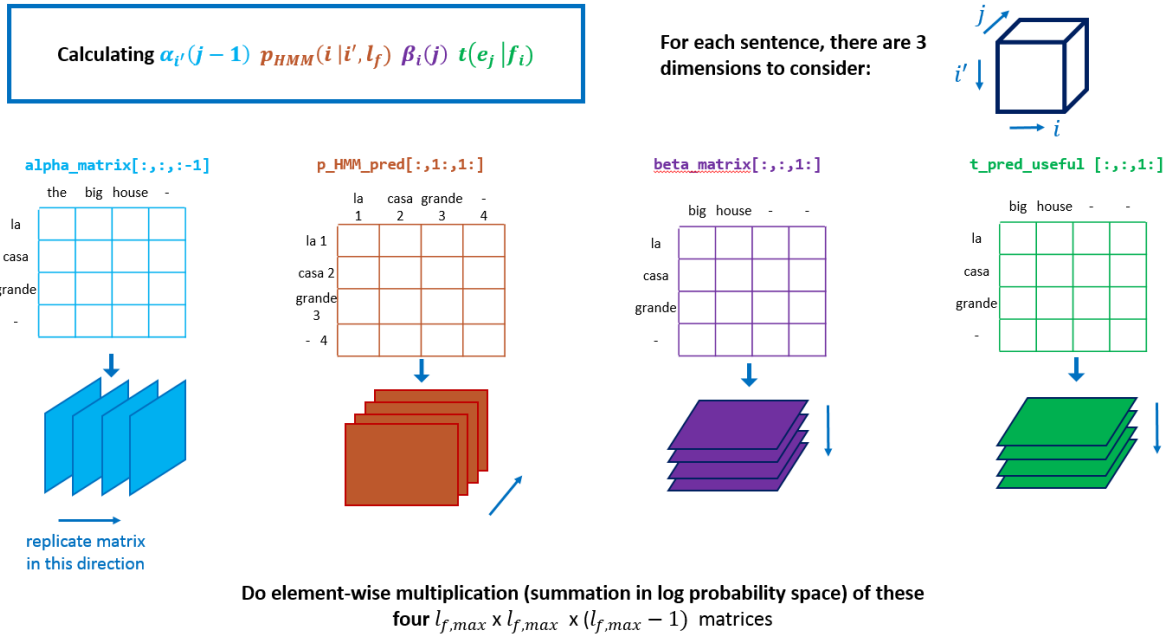


Fig. 3.4 Calculation of $a_{i'}(j-1)a(i \mid i')\beta_i(j)t(e_j \mid f_i)$ before summation over i', i, j.

Chapter 4

Experimental techniques & results

To restrict the search space, all neural models had one hidden layer. Continuing with notation introduced in chapter 2, embedding and hidden layers sizes are denoted by $D_{embed,M1}$, H_{M1}; $D_{embed,M2}$, H_{M2}; $D_{embed,HMM}$, H_{HMM} for the lexical translation probability, neural IBM Model 2 alignment probability and neural HMM alignment probability neural networks.

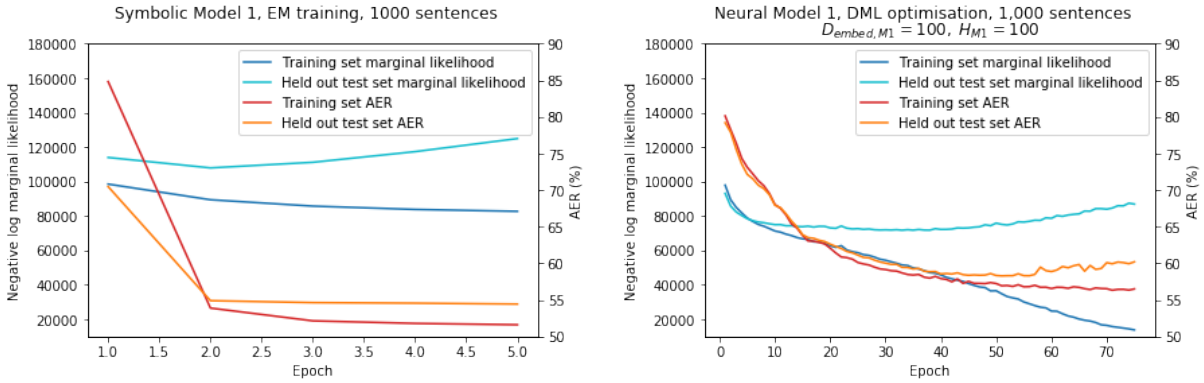
All of the following models were trained on the same 50,000 sentences in the SciELO dataset (apart from where stated otherwise).

4.1 Symbolic IBM Model 1, comparison with neural Model 1

The symbolic IBM Model 1 which was implemented in Python at the start of the project was trained on the largest reasonable dataset: 1000 sentences were used to get the balance of a large amount of data (hence, most likely, a better AER) and short enough computing time. The training time for 1000 sentences was 8.5 hours. The results for both are shown in Figure 4.1.

The 1000-sentence training set used here is the same as the 1000-sentence 'training subset' in all other experiments, allowing for direct comparison of AER between all models. The plotted 'negative log marginal likelihood values' can be compared directly within the same model family (i.e. within IBM Model 1, Model 2 and HMM results). This is because the likelihood values plotted for the IBM Model 1 and Model 2 are unnormalised. For the IBM Model 1, the marginal likelihood plotted is in fact the value $\prod_{j=1}^{l_e}\sum_{i=0}^{l_f} t(e_j \mid f_i)$ (cf. equation (2.10)). The normalising constants $\frac{\varepsilon}{(l_f+1)^{l_e}}$ are the same across the same set of sentences, hence direct comparisons can be made across all IBM Model 1 results.

The learning rate used for neural model training is 10^{-3}, which is the same for all other neural models in this section, unless otherwise stated. The symbolic IBM Model 1 shows classic EM behaviour, converging within just a few epochs. Although this beats the the neural IBM Model 1 in terms of number of epochs to converge (2 epochs compared to 40), the neural IBM Model 1 converged much faster in terms of time taken: training and computing likelihood and



(a) Symbolic IBM Model 1 trained by EM

(b) Neural IBM Model 1 trained by DML

Fig. 4.1 Evolution of marginal likelihood and AER for symbolic and neural IBM Model 1 trained on 1,000 sentences.

AER took almost 2 hours per epoch for the symbolic model, whereas all 75 epochs of the neural model were completed within 9 minutes.

Both graphs show overfitting: after a few epochs, the marginal likelihood of the held out test set begins to increase for both models. The AER of the held out test set also gets worse for the neural model, although it remains approximately constant for the symbolic model.

It is interesting that for the neural model, the marginal likelihood of the training set continues to decrease noticeably even after the training set AER has converged, highlighting the fact that the desired quantity (AER) does not necessary track the training criteria (marginal likelihood).

4.2 Neural IBM Model 1

4.2.1 DML

Combinations of different embedding and hidden layer sizes were investigated for the neural IBM Model 1 trained by DML. The values chosen for testing were $D_{embed,M1} \in [50, 100, 500]$ and hidden layer size $H_{M1} \in [50, 100, 500]$. The results are shown in Figure 4.2. The best performing model architecture was $D_{embed,M1} = 100$, $H_{M1} = 500$, which had an average AER of 39.00% in the last 10 epochs. To reduce computation time, the slightly smaller but reasonably-performing architecture of $D_{embed,M1} = 100$, $H_{M1} = 100$ was chosen for the lexical translation model of future experiments. This architecture's results are in the central panel of Figure 4.2.

We can see in the plots that for this amount of training data, 50 epochs were sufficient for both the AER and the marginal likelihood to converge.

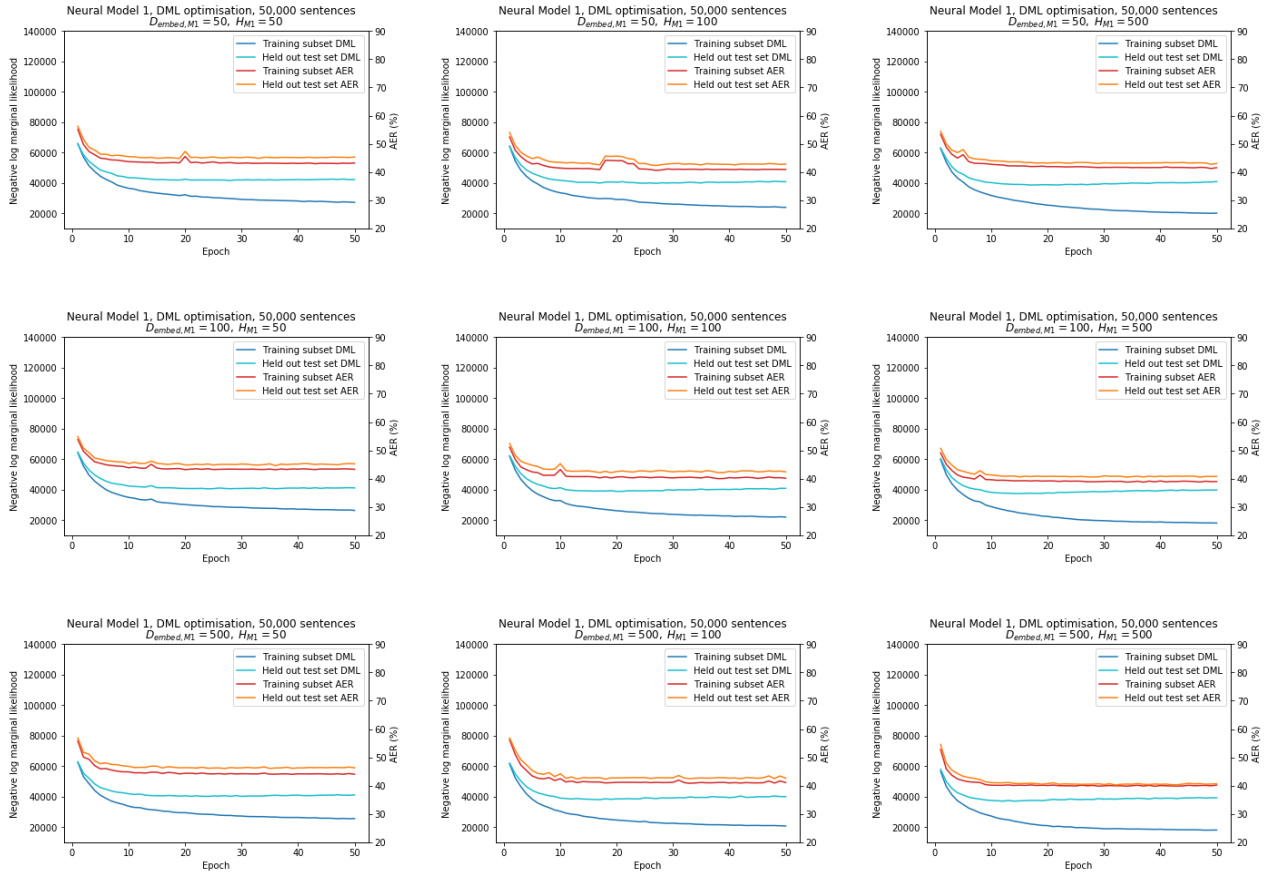


Fig. 4.2 Evolution of marginal likelihood and AER of neural IBM Model 1 trained by DML optimisation

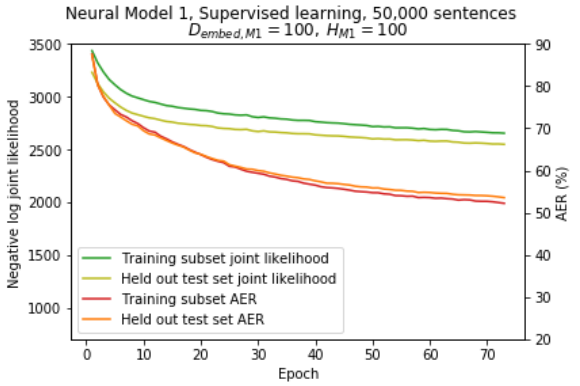
4.2.2 Viterbi training

Viterbi training of the neural IBM Model 1 was carried out, with the neural network initialised randomly. The results are shown in Figure 4.3. Although the negative log marginal likelihood is not the variable used to train the model, it decreases anyway.

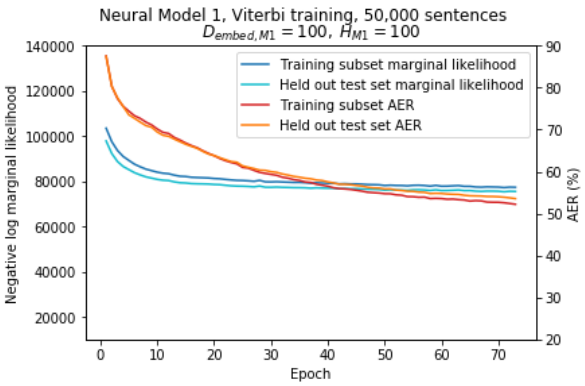
4.2.3 Supervised learning

Supervised training of the neural IBM Model 1 was carried out, with training data obtained from GIZA++ alignments. In order to speed up convergence, a learning rate of 10^{-2} was used, hence the slightly more bumpy graphs. Results are shown in Figure 4.4. Again, we see a similar trend were the marginal likelihood improves despite it not being the training criterion.

The final training set AER is 30.1% (averaged across the last 5 epochs), significantly lower than the previous, unsupervised models.

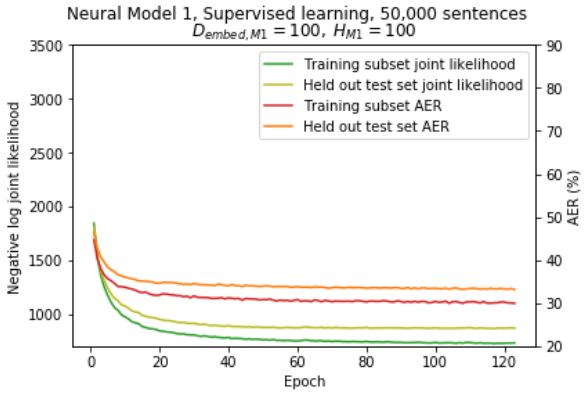


(a) Evolution of joint probability ($p(\mathbf{e}, \hat{a} \mid \mathbf{f})$, the criterion the model was trained on) and AER

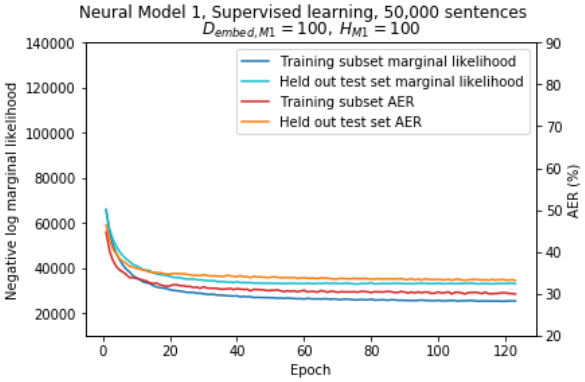


(b) Evolution of marginal likelihood ($p(\mathbf{e} \mid \mathbf{f})$, not a criterion the model was trained on) and AER

Fig. 4.3 Evolution of joint probability, marginal likelihood and AER during supervised neural Model 1 training.



(a) Evolution of joint probability ($p(\mathbf{e}, \hat{a} \mid f)$, the criterion the model was trained on) and AER



(b) Evolution of marginal likelihood ($p(\mathbf{e} \mid f)$, not a criterion the model was trained on) and AER

Fig. 4.4 Evolution of joint probability, marginal likelihood and AER during supervised neural Model 1 training.

4.3 Neural IBM Model 2

4.3.1 DML

The neural IBM Model 2 was trained with DML optimisation, with both neural networks initialised randomly. The marginal likelihood and AER plots are shown in Figure 4.5. Strictly speaking, the 'negative log marginal likelihood' plotted is offset from the true value: the metric being plotted is in fact $\prod_{j=1}^{l_e} \sum_{i=0}^{l_f} t(e_j \mid f_i)\, p_{M2}(i \mid j, l_e, l_f)$, without a normalising constant (cf. equation (2.18)).

The lexical translation probability neural network had parameters $D_{embed,M1} = 100$, $H_{M1} = 100$, and the alignment probability network had parameters $D_{embed,M2} = 100$, $H_{M2} = 50$. The learning rate for the alignment probability network parameters was set to 10^{-4} (and remained at 10^{-3} for the other network). Informal experimentation with different combinations of these

hyperparameters suggested that training is quite sensitive to them: if the alignment probability network was too large or had too large a learning rate, it would not converge to a low AER. As

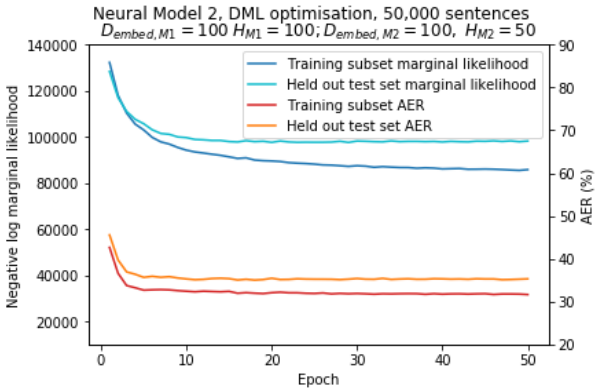
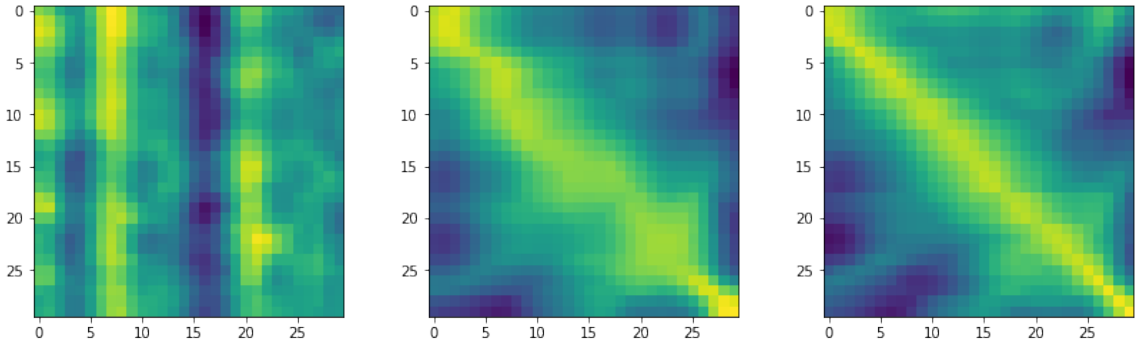


Fig. 4.5 Evolution of marginal likelihood and AER during neural IBM Model 2 training by DML optimisation

expected, the final AER of the neural IBM Model 2 (31.76%) is lower than the simpler neural IBM Model 1.

For interest, Figure 4.6 shows the outputs of the alignment probability matrix for a neural network trained on a smaller dataset (10,000 sentences with $l_f \leq 30$, $l_e \leq 30$).



(a) After 1st epoch (b) After 25th epoch (c) After 50th epoch

Fig. 4.6 Output of the alignment probability network (of neural IBM Model 2) for a sentence with $l_f = l_e = 30$ over the course of model training. This output will apply to any sentences with those lengths. Lighter areas have a higher probability mass. Spanish sentence indices i are along the vertical axis and English word indices j are along the horizontal axis (cf. Figure 2.3).

4.4 Neural HMM

4.4.1 DML

The results of a neural HMM trained by DML optimisation are shown in Figure 4.7. The parameters of both neural networks were initialised randomly. The marginal likelihood here is

the normalised value, obtained by equation (2.40). The parameters used here and in the following section are similar to the previous section: $D_{embed,M1} = 100$, $H_{M1} = 100$ and $D_{embed,HMM} = 100$, $H_{HMM} = 50$, with learning rates of 10^{-3} and 10^{-4} for the parameters of the respective models. As we would expect, the best AER of the neural HMM (37.35%) outperforms the AER of the best (unsupervised) neural IBM Model 1. The Viterbi probability matrix predicted by the

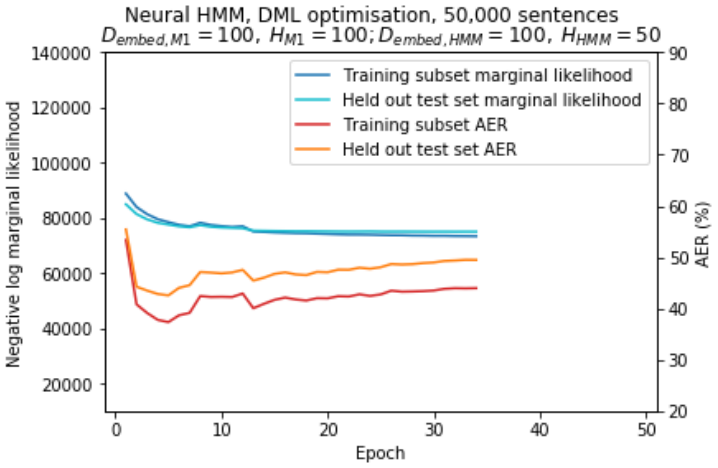


Fig. 4.7 Evolution of marginal likelihood and AER during neural HMM training by DML optimisation. The model was run for less than the usual 50 epochs due to the longer training time

neural HMM after the last epoch of DML training (the 34th epoch in this case) for the sentence pair 'all ethical aspects were considered' and 'todos los aspectos éticos fueron contemplados' is shown in Figure 4.8.

The outputs of both neural networks for the same sentence pair (and also after the final epoch) are shown in Figures 4.9.

4.4.2 EM

The evolution of the marginal likelihood and AER of a neural HMM trained by Generalised EM are shown in Figure 4.10. Again, the parameters of both neural networks were initialised randomly. As discussed in sections 2.1.3 and 3.7.2, the fact that we update the posterior on a batch-by-batch rather than epoch-by-epoch basis means that there is no guarantee that the likelihood of the training data will increase. However, the model is able to learn reasonable alignments with a minimum AER of 37.91 %, and the marginal likelihood of the training subset does increase.

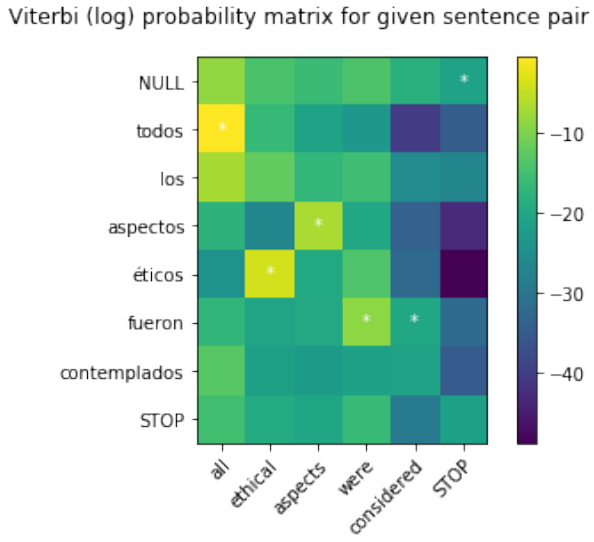
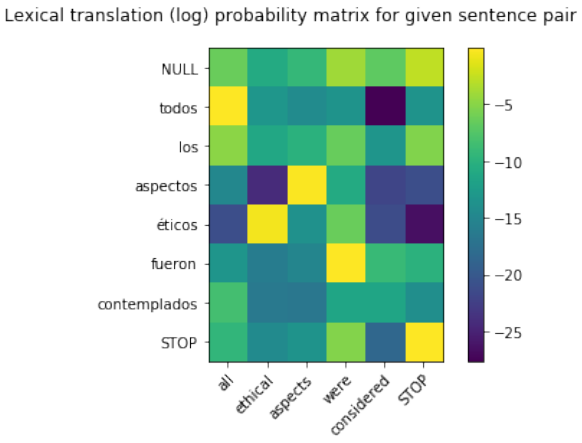
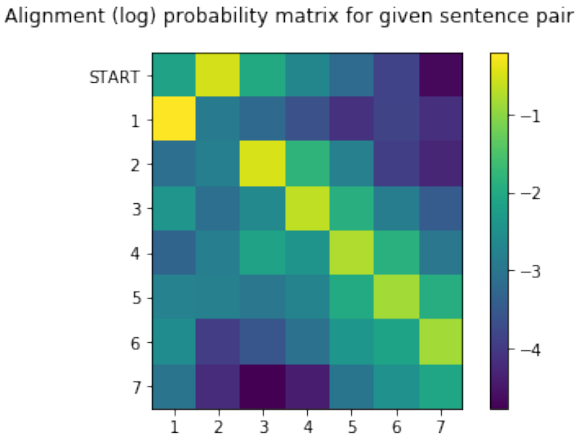


Fig. 4.8 Viterbi probability matrix for the sentence pair ‘all ethical aspects were considered’ and ‘todos los aspectos éticos fueron contemplados’. Lighter colours indicate a higher probability mass. A white asterisk (‘*’) is drawn on the alignment predictions (i.e. the maximum probability in each column).



(a) `t_pred_useful` (cf. Figure 3.1), output of lexical translation probability network for English words that are in the sentence in question.



(b) `p_HMM_pred` (cf. Figure 3.2), output of alignment probability network. i indices (the ‘from’ state) and i' indices (the ‘to’ state) are on the vertical and horizontal axes respectively

Fig. 4.9 Visualisation of output of neural HMM networks for the sentence pair ‘all ethical aspects were considered’ and ‘todos los aspectos éticos fueron contemplados’. Lighter colours indicate a higher probability mass.

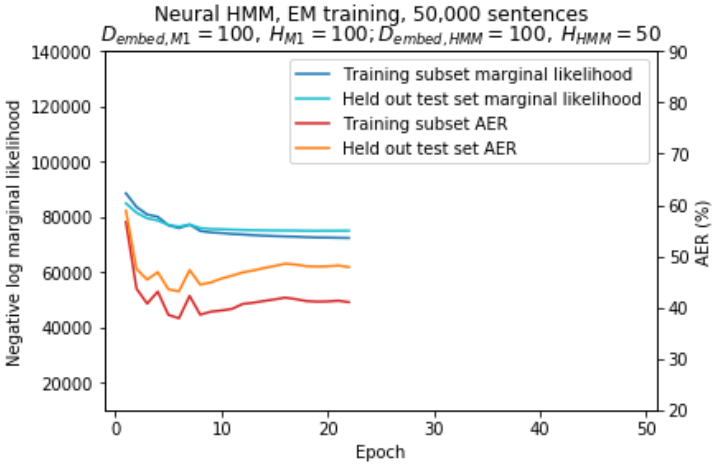


Fig. 4.10 Evolution of the marginal likelihood and AER during EM training of a neural HMM.

Chapter 5

Discussion

5.1 Symbolic IBM Model 1, comparison with neural Model 1

As the only experiment comparing the performance of a symbolic alignment model with its neural counterpart, it is interesting to note that the symbolic IBM Model 1 outperformed the neural IBM Model 2 with an AER that was lower by 5 %. This is perhaps not a fair comparison, as the models may have different relative performances with different amounts of data and, assuming that more data improves performance, the neural model has the advantage of being much quicker to train, even on large amounts of data. It is interesting to note that whereas convergence of the symbolic Model 1 took 3.5 hours, the neural model reached convergence within 5 minutes.

Furthermore, as no methodical search for the best neural architecture for this training set size was undertaken, it is possible that the neural model can achieve a better AER. This itself brings up an advantage that the symbolic IBM Model 1 has over the neural model: it requires no hyperparameter tuning, at least in this basic setup.

5.2 Neural IBM Model 1

5.2.1 DML

It is interesting to note that the variation in performance of the different architectures investigated was quite small: the converged training subset AER values spanned from 39.00% (for $D_{embed,M1} = 500, H_{M1} = 100$) to 44.13 % - a range of just over 5%.

As the best-performing neural architecture is on the edge of our search space, it is possible that the AER may be improved further by training on even larger networks, but as that would also increase training time, and the focus of this project was comparing different kinds of models, larger networks were not tested.

5.2.2 Viterbi training

As Viterbi training is 'hard' EM, one would expect Viterbi training to converge slower than EM. One would also expect expect EM to converge slower than DML (as EM is an indirect way of maximising likelihood), so it is not surprising that Viterbi is slower to converge than DML. It is also not surprising that the performance of Viterbi training is worse, both in terms of AER and marginal likelihood, as Viterbi training may get stuck in a local optimum, depending on the initialisation.

5.2.3 Supervised learning

The supervised learning results give an indication of the best performance we can expect from this network from a good enough signal. Potentially the AER could be reduced even further with a larger network, but we would expect a considerable error rate to remain due to inherent insufficiencies in the IBM Model 1, such as its inability to decide on the best alignment if the same word pair occurs more than twice in a sentence: for example, in the sentence pair 'the cat and the dog' & 'el gato y el perro', when picking alignments for 'the', the IBM Model 1 cannot distinguish between the first and second 'el'.

5.3 Neural IBM Model 2

5.3.1 DML

It is reassuring to see a clear performance boost in the neural IBM Model 2 (Figure 4.5) compared to the IBM Model 1: even though it is unsupervised, the neural IBM Model 2 does almost as well as the supervised neural IBM Model 1.

The results in Figure 4.6 show how the alignment probability model learns that words in English tend to be aligned to words in Spanish that are in a similar location in the sentence, hence the diagonal structure that emerges. It is interesting to note that the lower left corner is quite distinct, as punctuation marks at the end of sentences should be aligned with each other.

5.4 Neural HMM

It is noticeable that although the negative log marginal likelihood follows a general trend of decreasing and then converging, the AER falls and then rises. This is another instance when the behaviour of the desired quality does not track the training criteria. The optimal AER values occur at around the 5th epoch. This is consistent with symbolic HMM training with EM, for which normally only 5 EM iterations are carried out [5].

5.4.1 DML

It is not surprising that the neural HMM performs better than neural IBM 1. We would expect the neural HMM to perform better than the neural IBM Model 2 as that tends to be the case for the symbolic models [15]. The neural HMM performs slightly worse than the neural IBM Model 2 in these experiments. This may be due to the hyperparameters being suboptimal for the HMM, or perhaps it has got stuck in a local minimum during training.

The plots in Figures 4.8 and 4.9 provide an interesting insight into what the model learns. The alignments for the first 4 English words (‘all’, ‘ethical’, ‘aspects’ and ‘were’) are correct. The word ‘considered’ should be aligned to ‘contemplados’ (and it very nearly is, as the Viterbi probability in location (‘fueron’, ‘considered’) has almost the same value as the probability in location (‘considered’, ‘contemplados’)). This is not particularly surprising as the word ‘contemplados’ is a participle, which must be preceded by an auxiliary verb such as ‘fueron’. If these two words commonly co-occur in , then it will be difficult for the lexical translation probability to decide whether ‘fueron’ or ‘contemplados’ is a translation of ‘considered’. Indeed, we can see that the lexical translation probabilities for these two possible word pairs are very similar in Figure 4.9a.

It seems that the model has made a similar mistake by aligning the full stop with the ‘NULL’ token. Because the full stop is present in most of the sentences in the corpus, the ‘NULL’ token and full stop will co-occur very frequently.

As expected, there is a noticeable diagonal structure in Figure 4.9b as we would generally expect the latent variable i to make small ‘jumps’ between English word positions j.

Finally, it is worth pointing out again that Figures 4.8 and 4.9 are derived from the HMM model at the end of epoch 34 in Figure 4.7. It is quite possible the plots from around the 5th epochs would be more accurate.

5.4.2 EM

It is clear in Figure 4.10 that the negative log marginal likelihood decreases during training, even though we are not directly optimising for it.

It was mentioned in by Tran et al. [20] that neural HMMs for part of speech tagging trained by EM took longer to converge and performed slightly worse than neural HMMs trained by DML. For word alignment, the neural HMM trained by DML does have a slightly lower best AER, but by a very small margin. It is also difficult to comment on the convergence of the models, as the two examples we have seem to converge to a worse AER than at their peak. Thus there is not sufficient evidence to concluded whether the comment about performance and accuracy of DML and EM applies to neural HMMs trained for word alignment.

Chapter 6

Conclusion

Over the course of this project, neural versions of IBM Model 1, IBM Model 2 and the HMM were developed and trained using different methods: Direct Marginal Likelihood, Expectation Maximisation, Viterbi training and supervised learning.

Positional encodings were used to model the alignment probability distribution which allowed us to consider alignment probabilities in terms of the absolute positions of words rather than their relative positions, as is normally the case.

This use of positional encodings has allowed us to intuitively see the model learning the structure of alignment probabilities. As we have used English and Spanish which have quite similar word order grammar rules, the IBM Model 2 and HMM learnt to produce diagonal alignment probability distributions. It would be interesting to apply these models to languages that have a less similar word order to English, such as Japanese and Mandarin.

The best AERs of the neural IBM Model 1, IBM Model 2 and HMM were 39.00%, 31.76% and 37.35% respectively. As expected, both the IBM Model 2 and the HMM outperform the IBM Model 1, although one would also expect the HMM to outperform the IBM Model 2. However, this report explored a very narrow set of possible hyperparameters, so it is difficult to draw conclusions about the relative performance of these neural models, and it would be interesting to do a more comprehensive hyperparameter search to better understand how accurate each family of models can be.

References

[1] Baum, L. E. (1972). An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes. In Shisha, O., editor, *Inequalities III: Proceedings of the Third Symposium on Inequalities*, pages 1–8, University of California, Los Angeles. Academic Press.

[2] Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*.

[3] Brown, P. F., Pietra, V. J. D., Pietra, S. A. D., and Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Comput. Linguist.*, 19(2):263–311.

[4] Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38.

[5] Deng, Y. and Byrne, W. (2008). Hmm word and phrase alignment for statistical machine translation. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(3):494–507.

[6] Devlin, J., Zbib, R., Huang, Z., Lamar, T., Schwartz, R., and Makhoul, J. (2014). Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1370–1380, Baltimore, Maryland. Association for Computational Linguistics.

[7] Gales, M. (2018). 4f10: Variational autoencoder. University Lecture.

[8] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

[9] Goldberg, Y. (2016). A primer on neural network models for natural language processing. *J. Artif. Int. Res.*, 57(1):345–420.

[10] Guta, A., Alkhouli, T., Peter, J.-T., Wuebker, J., and Ney, H. (2015). A comparison between count and neural network models based on joint translation and reordering sequences. In *EMNLP*.

[11] Janocha, K. and Czarnecki, W. (2017). On loss functions for deep neural networks in classification. *CoRR*, abs/1702.05659.

[12] Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[13] Koehn, P. (2010). *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA, 1st edition.

[14] Och, F. J. and Ney, H. (2000). Improved statistical alignment models. In *ACL*.

[15] Och, F. J. and Ney, H. (2003). A systematic comparison of various statistical alignment models. *Comput. Linguist.*, 29(1):19–51.

[16] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch. In *NIPS-W*.

[17] Ramachandran, P., Zoph, B., and Le, Q. V. (2018). Searching for activation functions. *CoRR*, abs/1710.05941.

[18] Robbins, H. and Monro, S. (1951). A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407.

[19] Sennrich, R., Haddow, B., and Birch, A. (2016). Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909.

[20] Tran, K., Bisk, Y., Vaswani, A., Marcu, D., and Knight, K. (2016). Unsupervised neural hidden markov models. *CoRR*, abs/1609.09007.

[21] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

[22] Vogel, S., Ney, H., and Tillmann, C. (1996). Hmm-based word alignment in statistical translation. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 2*, COLING '96, pages 836–841, Stroudsburg, PA, USA. Association for Computational Linguistics.

[23] Wang, W., Alkhouli, T., Zhu, D., and Ney, H. (2017). Hybrid neural network alignment and lexicon model in direct HMM for statistical machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 125–131, Vancouver, Canada. Association for Computational Linguistics.

[24] Yang, N., Liu, S., Li, M., Zhou, M., and Yu, N. (2013). Word alignment modeling with context dependent deep neural network. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 166–175, Sofia, Bulgaria. Association for Computational Linguistics.

Appendix

Acknowledgements

I wish to thank my supervisor, Prof. Bill Byrne for his guidance with this project, and Felix Stahlberg and Danielle Saunders for their suggestions for improving the model's speed and their help with acquiring language data.

Risk assessment retrospective

As this is a computer-based project, no hazards were anticipated, and none were encountered.