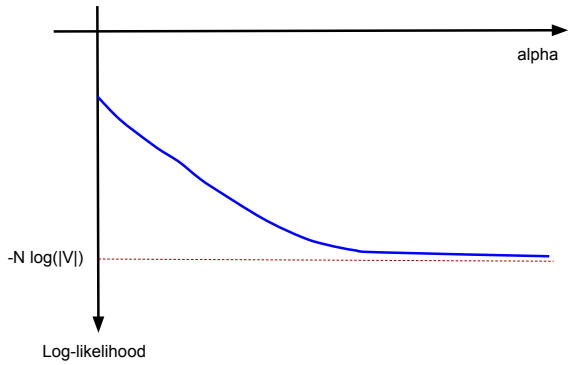
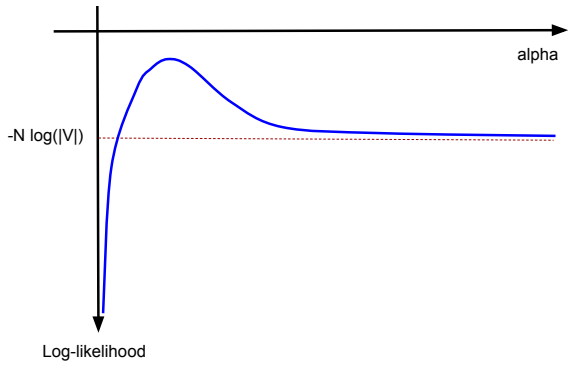
1: Smoothing

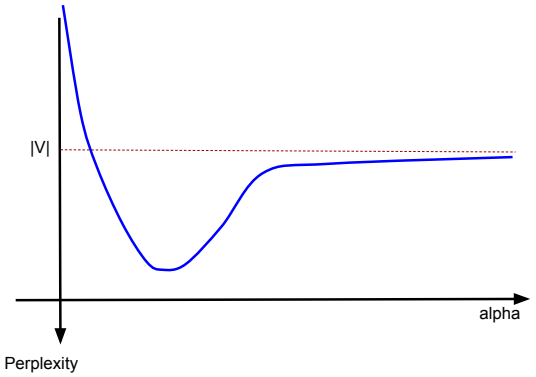
1. When using add-α smoothing, larger values of α will spread the probability mass away from the maximum training log-likelihood solution. As a result, the training log-likelihood will decrease (get worse) as α increases.



2. If the test corpus contains a word not in the training set, then setting $\alpha = 0$ will cause the test log-likelihood to be $-\inf$. This is because the model would assign probability zero to this new word. Increasing α will again spread more of the probability mass over all the words, in any context. This will initially help and the test log-likelihood will increase. However, let's look at what happens to the trigram probabilities when α is very large. In this case, $\lim_{\alpha\to+\infty} P(w_i|w_{i-2}, w_{i-1}) = \lim_{\alpha\to+\infty} \frac{\text{count}(w_{i-2},w_{i-1},w_i)+\alpha}{\text{count}(w_{i-2},w_{i-1})+\alpha|V|} = \frac{1}{|V|}$. In other words, the trigram turns into a uniform distribution over the words. This is unlikely to be very good in terms of test log-likelihood. Taken together, we would expect the test log-likelihood to behave as follows



3. Since Perplexity $= 2^{-\frac{1}{N}l_{\text{corpus}}}$, we can plot the perplexity on the test corpus directly from the previous question.



4. $\lim_{\alpha\to+\infty} \text{Perplexity} = \lim_{\alpha\to+\infty} 2^{-\frac{1}{N}\text{l}_{\text{corpus}}} = \lim_{\alpha\to+\infty} 2^{-\frac{1}{N}N\log\left(\frac{1}{|V|}\right)} = |V|$

As $\alpha \to +\inf$, the perplexity tends to $|V|$, which is the perplexity of the uniform trigram probabilities (see the notes of lecture 2).

2: Neural Language Models

1. We count the number of scalar operations needed for each step:

- Assume obtaining each x_i is $O(1)$ so setting x takes $O(nd)$ operations for an $n-$gram model.
- To compute each z_j^h it takes $O(nd)$, hence to compute all z_j^h it takes $O(dm)$.
- To compute each f_j^h it takes $O(1)$, hence to compute all f_j^h it takes $O(m)$.
- To compute each z_k^o it takes $O(m)$, hence to compute all z_k^o it takes $O(m|V|)$.
- We need an additional $O(|V|)$ operations to evaluate p_k, $k = 1, \ldots, |V|$, once z_k^o, $k = 1, \ldots, |V|$ are available.

As a result, the number of scalar operations needed to perform one forward pass through the network is $O(nd) + O(ndm) + O(m) + O(m|V|) + O(|V|)$, i.e. $O(m(|V| + nd))$

2. If we can process each unit of the same layer in parallel:

- Storing x in the input units is now $O(1)$ (one step)
- Evaluating each z_j^h takes $O(nd)$, hence to compute all z_j^h will now also take $O(nd)$.
- Each f_j^h takes $O(1)$ once we have z_j^h so evaluating all of them in parallel takes $O(1)$ time.
- Each z_k^o takes $O(m)$, and thus all in parallel take $O(m)$.
- Each $\exp(z_k^o)$ takes $O(1)$ time so all of them can be evaluated in $O(1)$ in parallel. We still need to sum these to obtain the normalization constant. If we arrange the terms hierarchically (binary tree), the sum can be implemented in parallel in $O(\log|V|)$ steps. Once the sum is available, $p_k = \exp(z_k^o)/\text{sum}$ can be evaluated in parallel in one step.

As a result, the number of parallel steps needed to perform one forward pass is $O(1) + O(nd) + O(1) + O(m) + O(\log |V|)$, i.e. $O(\log |V| + m + nd)$. This would make the model feasible even for a reasonably large corpus.

3.

$$\delta_k^o = \frac{\partial \log p_y}{\partial z_k^o} = \frac{\partial}{\partial z_k^o}\left(z_y^o - \log\left(\sum_{l=1}^{|V|} \exp(z_l^o)\right)\right) = \frac{\partial z_y^o}{\partial z_k^o} - \frac{\exp(z_o^k)}{\sum_{l=1}^{|V|} \exp(z_o^l)} = \frac{\partial z_y^o}{\partial z_k^o} - p_k$$

Hence:

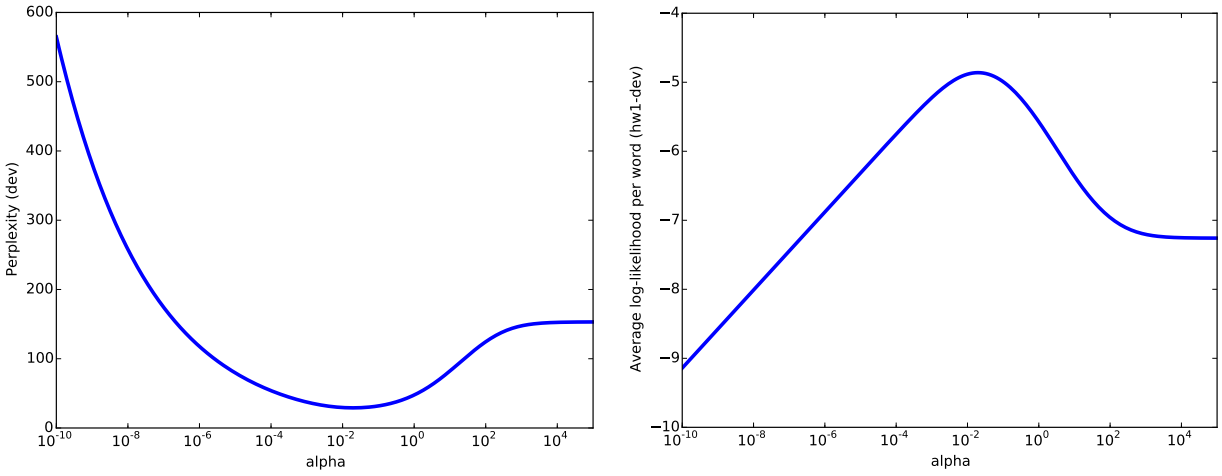
$$\delta_k^o = \begin{cases} 1 - p_k & \text{if } y = k \\ -p_k & \text{otherwise} \end{cases}$$

Regarding the other gradients:

$$\begin{aligned} \delta_j^h &= \frac{\partial \log p_y}{\partial z_j^h} = \frac{\partial f_j^h}{\partial z_j^h}\frac{\partial \log p_y}{\partial f_j^h} \\ &= \frac{\partial f_j^h}{\partial z_j^h}\sum_{k=1}^{|V|}\frac{\partial z_k^o}{\partial f_j^h}\frac{\partial \log p_y}{\partial z_k^o} = (1 - (f_j^h)^2)\sum_{k=1}^{|V|} W_{jk}^o \delta_k^o \\ \delta_i^x &= \frac{\partial \log p_y}{\partial x_i} = \sum_{j=1}^{m}\frac{\partial z_j^h}{\partial x_i}\frac{\partial \log p_y}{\partial z_j^h} = \sum_{j=1}^{m} W_{ij}^h \delta_j^h \end{aligned}$$

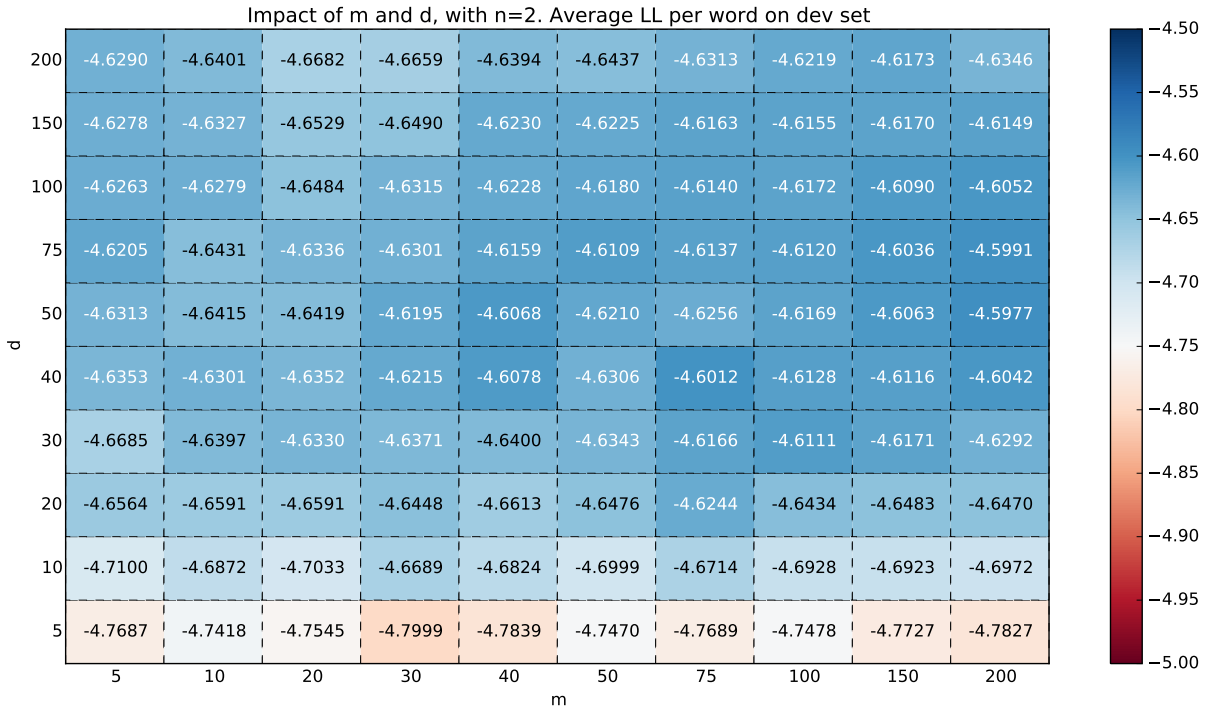
$$v(w_{t-2}) \leftarrow v(w_{t-2}) + \eta[\delta_1^x, \delta_2^x, ..., \delta_d^x]^T$$

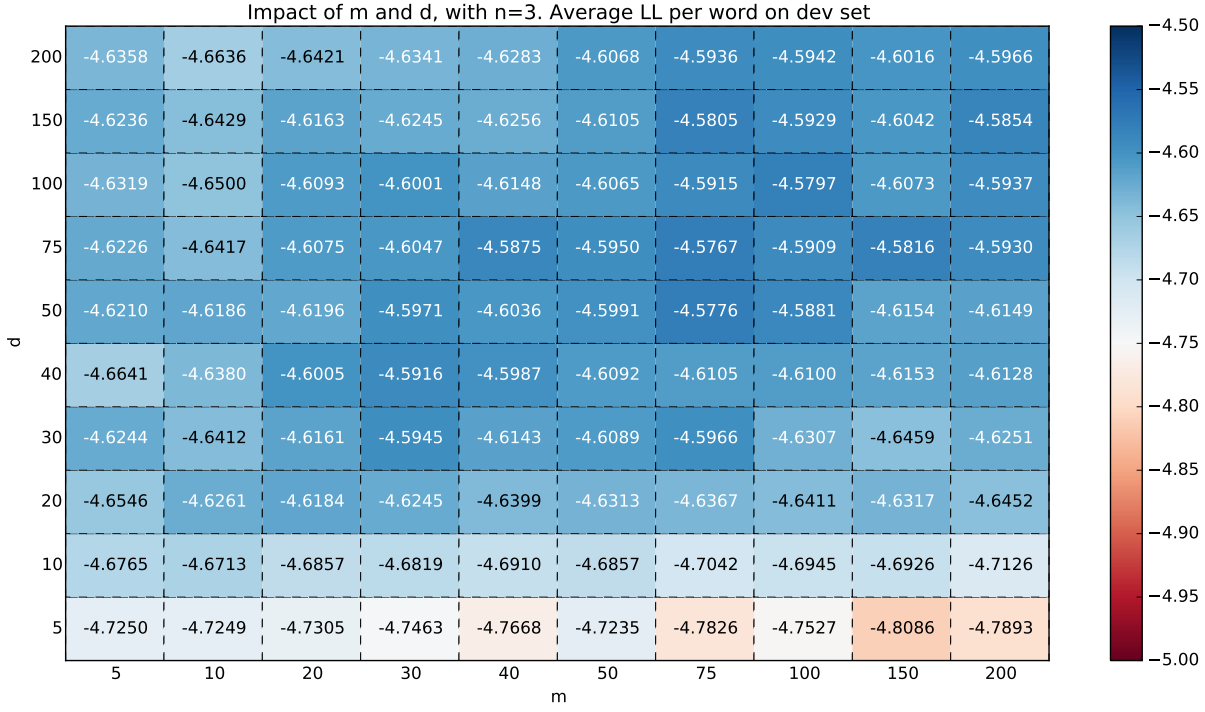
4a. To find the best α for the bigram model with add-α smoothing, we plot α against the perplexity and the average log-likelihood per word. For each α, the model is learnt using the training set, and the perplexity is computed on the dev set. The optimal α (i.e. the α that minimizes the perplexity) is approximately 0.022, as one can see in the two following graphs:

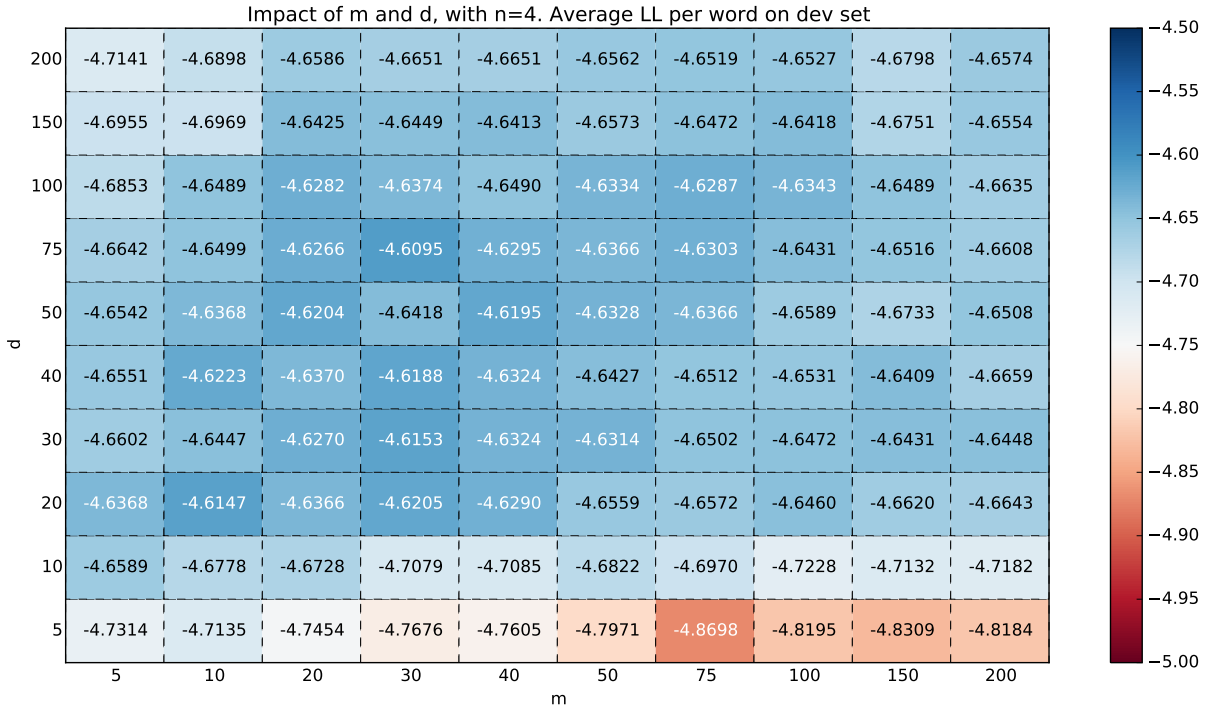


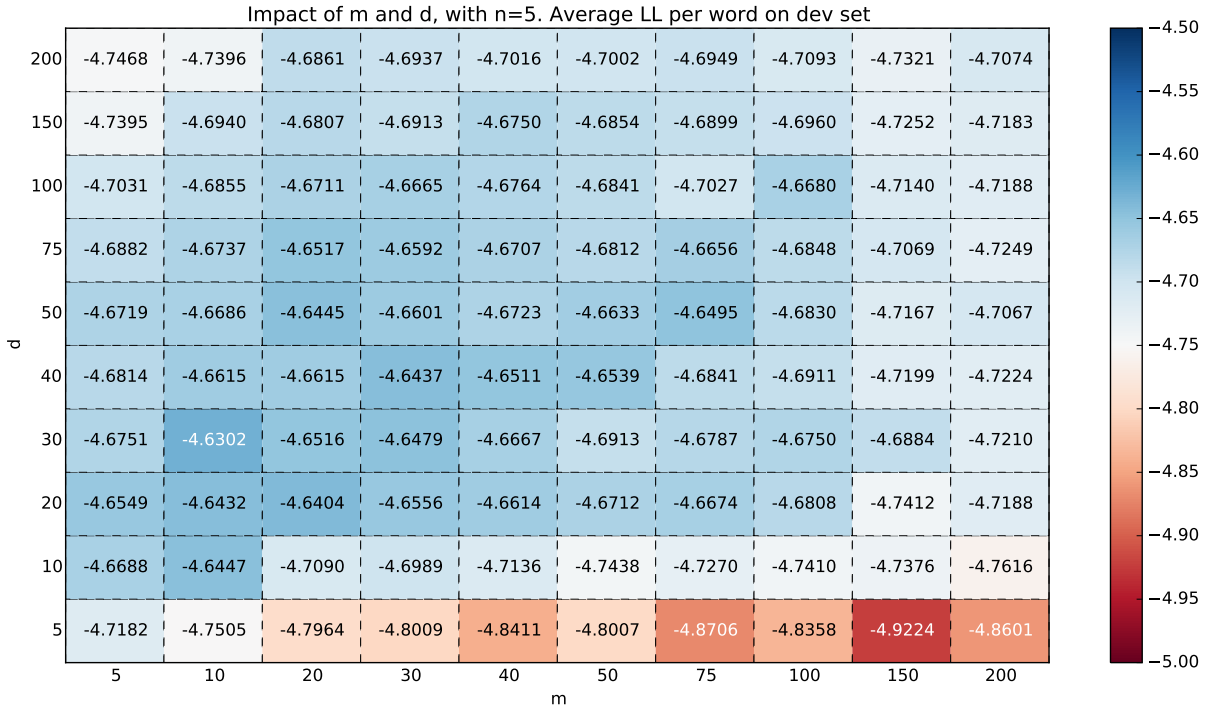
To find the best n, d, and m, we perform a grid search, and look at the average log-likelihood per word on the dev set at the last iteration of the training phase. The maximum number of

iterations is set to 50, but if after one iteration the average log-likelihood per word on the dev set hasn't increased, we stop the training. Note that this "early stopping" criterion will mitigate differences between configurations. The datasets are small so the dev set log-likelihood may increase quickly for complex models. Thus simpler models will be trained more, complex models less, washing out some of the differences.







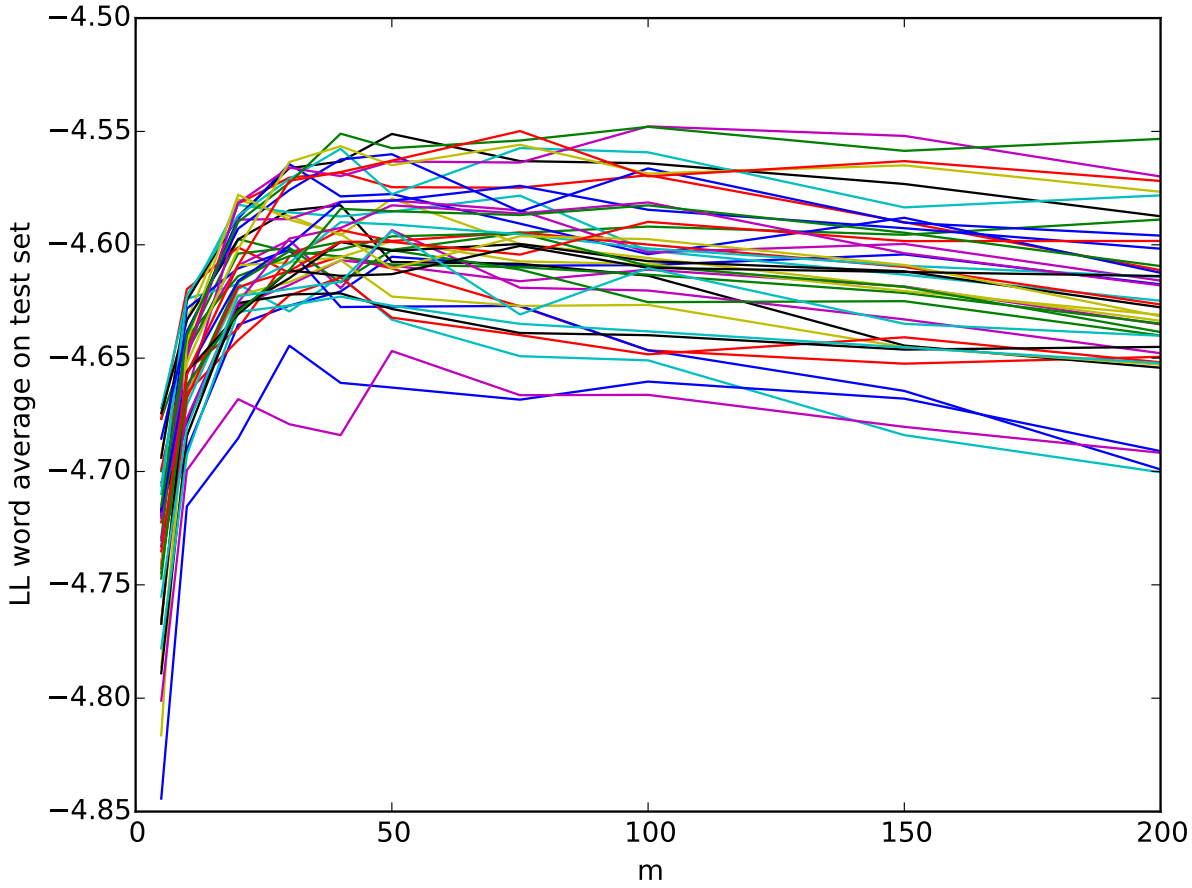


Looking at these results, we see that the best configuration is $n = 3$, $d = 75$, $m = 75$. Using this configuration, we get an average log-likelihood per word of -4.5637235 on the test set.

The bigram model with add-α smoothing with $\alpha = 0.022$ (the best configuration) achieves an average log-likelihood per word of -4.8455004583 on the test set.

4b. See previous answer.

4c. Increasing m from 1 to 25 significantly improves the model performance at the beginning, but when m reaches 50 the model performance's plateaus out, even slightly decreases. In the graph below, each line correspond to one specific configuration of n, m and d (same ranges as in the heatmaps above):



5a. The trained neural network model will assign zero or near zero probability to any word not seen in the training set regardless of what precedes it. Let k be a word that wasn't seen in the train set. During training, it will be advantageous for W^o_{0k} to approach $-\infty$ since otherwise the k^{th} word will take probability mass away from others. More precisely, consider W^o_{0k} update equation

$$W^o_{0k} := W^o_{0k} + \eta\delta^o_k = W^o_{0k} - \eta p_k \quad \text{(since word } k \text{ does not appear)}$$

Thus $W^o_{0k} \to -\infty$ until $p_k \to 0$.

5b. Yes, the trained model can provide a reasonable probability estimate for w_t even if the specific combination of the $n-1$ preceding words didn't appear in the training set. The assumption here is that all the words did appear in some combination in the training set so that their word vectors can be estimated. Moreover, each training ngram, regardless of the specific preceding words, provides evidence for how word vectors should be combined. The model is therefore well trained to assess how word vectors combine.

6. The bigram model with add-α smoothing with $\alpha = 0.022$ yields an average log-likelihood per word of -5.19939989734 (corpus log-likelihood: -41.5951991787) for the first sentence and -6.81707401904 (corpus log-likelihood: -54.5365921523) the second sentence.

The neural language model with the best configuration found above, viz. $n = 3$, $d = 75$, $m = 75$, yields an average log-likelihood per word of -4.84783127062 (corpus log-likelihood: -38.7826501649) for the first sentence and -5.42208843528 (corpus log-likelihood: -43.3767074823) the second sentence.

7(1). (N).

7(2). (Y).