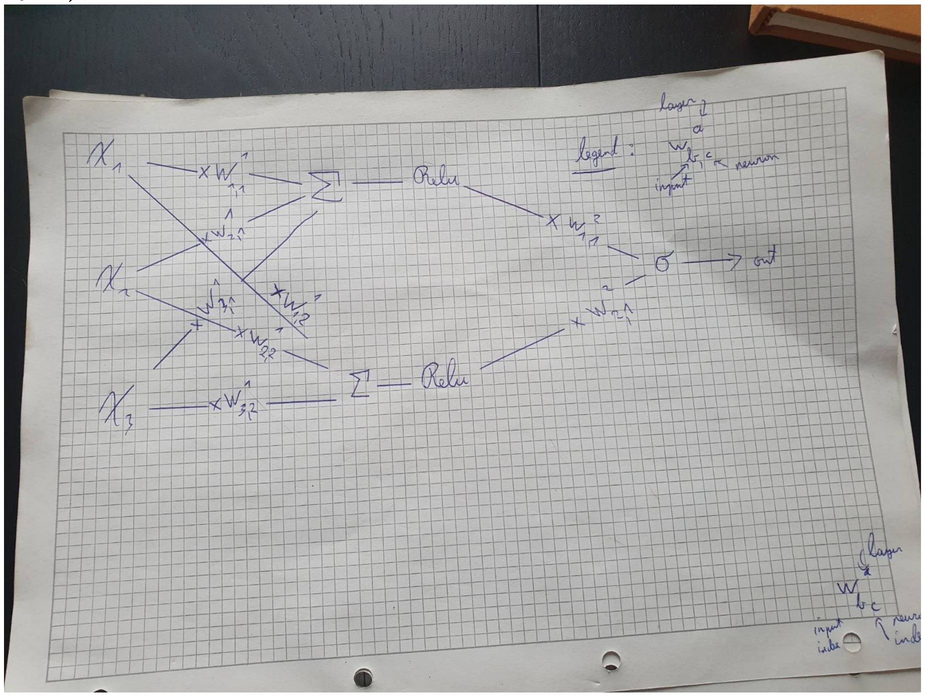
NLP assignment #1

Andrius Buinovskij - 18-940-270

Q1 a)



Q1 b) i)

Well since all inputs are 1 and all weights are 1 then

$$\mathbf{s}^1_1 = \sum_{i=1}^{3} \mathbf{x}_i \cdot \mathbf{w}^1_{i,1} = 3 \tag{1}$$

$$\mathbf{s}^1_2 = \sum_{i=1}^{3} \mathbf{x}_i \cdot \mathbf{w}^1_{i,2} = 3 \tag{2}$$

Where $\mathbf{s}^i_j$ is the sum input to the j'th neuron in the i'th layer, and $\mathbf{s}^i$ is a vector whose length is equal to the number of neurons in the i'th layer, with the input being the 0'th layer. So $\mathbf{s}^1$ is of length 2 since 1'st layer has 2 neurons.

Let $\mathbf{n}^i_j$ be the output of the j'th neuron in the i'th layer, then of course $\mathbf{n}^i$ is a vector of length equal to the number of neurons in the i'th layer:

$$n^i_j = ReLU\big(\mathbf{s}^i_j\big) \tag{3}$$

So in our case We get

$$\begin{align}\mathbf{n}^1_1 &= ReLU\big(\mathbf{s}^1_1\big) = ReLU\big(3\big) = 3 \tag{4}\\ \mathbf{n}^1_2 &= ReLU\big(\mathbf{s}^1_2\big) = ReLU\big(3\big) = 3 \tag{5}\end{align}$$

Same steps in the next layer:

$$\mathbf{s}^2_1 = 3 \cdot 1 + 3 \cdot 1 = 6 \tag{6}$$

And now We pass this through a sigmoid for our output instead of a *ReLU* so We get

$$out = \sigma\big(\mathbf{s}^2_1)\big) = \frac{1}{1+e^{-6}} = 0.99752737684 \tag{7}$$

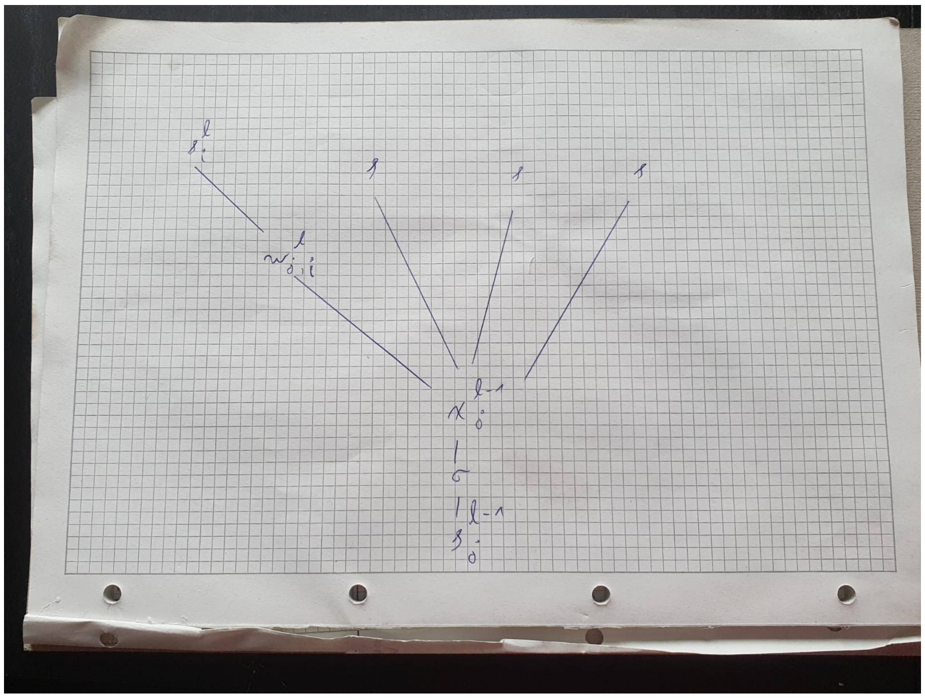
Q1 b) ii-iv)

Alright let's just roll all of these questions into one and do an iteration of backprop.

$$\begin{align}L_{BCE} &= -(y\log(\hat{y}) + (1-y)\log(1-\hat{y})) \tag{8}\\ &= -(0\cdot\log(0.99752737684) + (1-0)\log(1-0.99752737684)) \tag{9}\\ &= -(\log(0.00247262316)) \tag{10}\\ &= 2.60684206722 \tag{11}\end{align}$$

The forward pass is trivial.

Here is a pictorial view of the terms about to be defined:



Let $\mathbf{x}_i^l$ be the output of the i'th neuron in the l'th layer.

Let $\mathbf{w}_{i,j}^l$ be the weight belonging to j'th neuron multiplying the i'th input in the l'th layer. $\mathbf{w}_{\cdot,j}^l$ is then simply the weight vector associated with the j'th neuron in the l'th layer.

Let $\mathbf{s}_j^l$ be $(\mathbf{x}^{l-1})^\top \mathbf{w}_{\cdot,j}^l$, the weighted sum of inputs to the j'th neuron in the l'th layer.

We can then say that $x_i^l = \sigma(\mathbf{s}_j^l)$, where σ is the nonlinearity of choice.

Then define

$$\delta_j^l = \frac{\partial L_{BCE}}{\partial \mathbf{s}_j^l} \tag{12}$$

And finally

$$\delta_j^{l-1} = \sum_{i=1}^{d^l} \frac{\partial L_{BCE}}{\partial \mathbf{s}_i^l} \frac{\partial \mathbf{s}_i^l}{\partial \mathbf{x}_j^{l-1}} \frac{\partial \mathbf{x}_j^{l-1}}{\partial \mathbf{s}_j^{l-1}} \tag{13}$$

$$= \sum_{i=1}^{d^l} \delta_i^l \frac{\partial \mathbf{s}_i^l}{\partial \mathbf{x}_j^{l-1}} \frac{\partial \mathbf{x}_j^{l-1}}{\partial \mathbf{s}_j^{l-1}} \tag{14}$$

Where d^l is the number of neurons in layer l. So now We have a recursive definition which uses dynamic programming. This could be further nuanced by expressing things in matrix notation, but it's good enough for present purposes.

So now

$$\frac{\partial L_{BCE}}{\partial \mathbf{s}_1^2} = \frac{\partial L_{BCE}}{\partial \hat{y}} \frac{\partial \hat{y}}{\mathbf{s}_1^2} \tag{15}$$

$$\begin{aligned}
\frac{\partial L_{BCE}}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}} - \big(y \log(\hat{y}) + (1-y)\log(1-\hat{y})\big) && (16)\\
&= -y\frac{1}{\hat{y}} - (1-y)\frac{\partial}{\partial \hat{y}} \log(1-\hat{y}) && (17)\\
&= -y\frac{1}{\hat{y}} - (1-y)\frac{-1}{1-\hat{y}} && (18)\\
&= -y\frac{1}{\hat{y}} + \frac{1-y}{1-\hat{y}} && (19)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \hat{y}}{\mathbf{s}_1^2} &= \frac{\partial}{\mathbf{s}_1^2}\sigma(\mathbf{s}_1^2) && (20)\\
&= \sigma(\mathbf{s}_1^2) \cdot (1-\sigma(\mathbf{s}_1^2)) && (21)
\end{aligned}$$

So then We have

$$\begin{aligned}
\delta_1^2 = \frac{\partial L_{BCE}}{\partial \mathbf{s}_1^2} &= \frac{\partial L_{BCE}}{\partial \hat{y}} \frac{\partial \hat{y}}{\mathbf{s}_1^2} && (22)\\
&= \left(-y\frac{1}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)\left(\sigma(\mathbf{s}_1^2) \cdot (1-\sigma(\mathbf{s}_1^2))\right) && (23)
\end{aligned}$$

Alright. We only have one more of these to figure out:

$$\delta_j^1 = \sum_{i=1}^{d^2} \delta_i^2 \frac{\partial \mathbf{s}_i^2}{\partial \mathbf{x}_j^1} \frac{\partial \mathbf{x}_j^1}{\partial \mathbf{s}_j^1} \tag{24}$$

But since $d^2 = 1$, i.e. there is only one neuron in the output layer, We have

$$\begin{aligned}
\delta_j^1 &= \sum_{i=1}^{d^2} \delta_i^2 \frac{\partial \mathbf{s}_i^2}{\partial \mathbf{x}_j^1} \frac{\partial \mathbf{x}_j^1}{\partial \mathbf{s}_j^1} && (25)\\
&= \delta_1^2 \frac{\partial \mathbf{s}_1^2}{\partial \mathbf{x}_j^1} \frac{\partial \mathbf{x}_j^1}{\partial \mathbf{s}_j^1} && (26)\\
&= \delta_1^2 \mathbf{w}_j^2 \sigma(\mathbf{s}_j^1)(1-\sigma(\mathbf{s}_j^1)) && (27)
\end{aligned}$$

Now for the gradient update We will of course need

$$\frac{\partial L_{BCE}}{\partial \mathbf{w}^l_{i,j}} \tag{28}$$

But this can be easily derived since

$$\begin{align}
\frac{\partial L_{BCE}}{\partial \mathbf{w}^l_{i,j}} &= \frac{\partial L_{BCE}}{\partial \mathbf{s}^l_j}\frac{\partial \mathbf{s}^l_j}{\partial \mathbf{w}^l_{i,j}} \tag{29}\\
&= \delta^l_j \mathbf{x}^{l-1}_i \tag{30}
\end{align}$$

Now We do a forward pass and We know that $\mathbf{s}^1_i = 3$ and $\mathbf{s}^2_1 = 6$. Plugging in values We then get

$$\begin{align}
\delta^2_1 &= \Bigg(-y\frac{1}{\hat{y}} + \frac{1-y}{1-\hat{y}}\Bigg)\Bigg(\sigma(\mathbf{s}^2_1)\cdot(1-\sigma(\mathbf{s}^2_1))\Bigg) \tag{31}\\
&= \Bigg(\frac{1}{1-0.99752737684}\Bigg)\Bigg(\sigma(6)\cdot(1-\sigma(6))\Bigg) \tag{32}\\
&= \Bigg(\frac{1}{1-0.99752737684}\Bigg)\Bigg(0.99752737684\cdot(1-0.99752737684)\Bigg) \tag{33}\\
&= 404.428792942 \cdot 0.00246650929 \tag{34}\\
&= 0.99752737493 \tag{35}
\end{align}$$

Similarly We have

$$\begin{align}
\delta^1_j &= \delta^2_1 \mathbf{w}^2_j \sigma(\mathbf{s}^1_j)(1-\sigma(\mathbf{s}^1_j)) \tag{36}\\
\delta^1_1 &= \delta^2_1 \mathbf{w}^2_1 \sigma(\mathbf{s}^1_1)(1-\sigma(\mathbf{s}^1_1)) \tag{37}\\
&= 0.997527374931 \cdot \sigma(3)(1-\sigma(3)) \tag{38}\\
&= 0.997527374931 \cdot 0.95257412682 \cdot 0.04742587317 \tag{39}\\
&= 0.04506495478 \tag{40}
\end{align}$$

δ^1_2 is equal to δ^1_1.
Now that We have the deltas We can use

$$\frac{\partial L_{BCE}}{\partial \mathbf{w}^l_{i,j}} = \delta^l_j \mathbf{x}^{l-1}_i \tag{41}$$

For weights in the first layer this means

$$\begin{align}
\frac{\partial L_{BCE}}{\partial \mathbf{w}^1_{i,j}} &= \delta^1_j \mathbf{x}^0_i \tag{42}\\
&= 0.04506495478 \cdot 1 \tag{43}\\
&= 0.04506495478 \tag{44}
\end{align}$$

Where the 0'th layer is the input layer.

$$\frac{\partial L_{BCE}}{\partial \mathbf{w}^2_{i,j}} = \delta^2_j \mathbf{x}^1_i \quad (45)$$
$$= 0.99752737493 \cdot 3 \quad (46)$$
$$= 2.99258212479 \quad (47)$$

All that is left is to perform a step for each weight. All weights in the first layer simplify to

$$w^1_{i,j} = 1 - 0.1 \cdot 0.04506495478 \quad (48)$$
$$= 0.99549350452 \quad (49)$$

and similarly

$$w^2_{i,j} = 1 - 0.1 \cdot 2.99258212479 \quad (50)$$
$$= 0.70074178752 \quad (51)$$

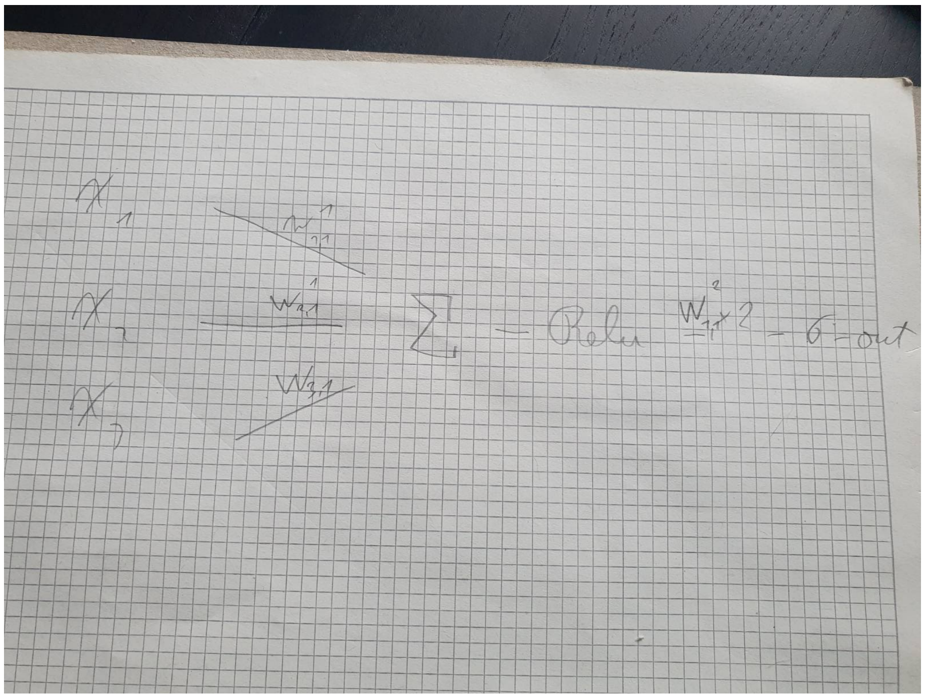
Sidenote
So when actually using this in the implementation of Q2, We get the following simplification:

$$\delta^2_1 = \frac{\partial L_{BCE}}{\partial \mathbf{s}^2_1} = \frac{\partial L_{BCE}}{\partial \hat{y}} \frac{\partial \hat{y}}{\mathbf{s}^2_1} \quad (52)$$
$$= \Big(-y\frac{1}{\hat{y}} + \frac{1-y}{1-\hat{y}} \Big) \Big(\sigma(\mathbf{s}^2_1) \cdot (1 - \sigma(\mathbf{s}^2_1)) \Big) \quad (53)$$
$$= \Big(-y\frac{1}{\hat{y}} + \frac{1-y}{1-\hat{y}} \Big) \Big(\hat{y} \cdot (1 - \hat{y}) \Big) \quad (54)$$
$$= -\hat{y} \cdot (1-\hat{y}) \cdot \frac{y}{\hat{y}} + \hat{y} \cdot (1-\hat{y}) \cdot \frac{1-y}{1-\hat{y}} \quad (55)$$
$$= -(1-\hat{y}) \cdot y + \hat{y} \cdot (1-y) \quad (56)$$

Q4 c))
Due to the fact that the weights were initialized as a uniform constant, I think that the second neuron in the hidden layer is redundant.

I don't think We can get rid of the hidden layer all together due to the ReLU - if We simply passed the output of the sum to the sigmoid, the sigmoid could receive negative terms, which is not possible under the current architecture. As it is, the input to the sigmoid is bounded to be above zero regardless of the weights.

As it stands, the below is certainly more efficient and has fewer parameters:



The reason the $\times 2$ is there is to make the computation identical regardless of what the weights are initialized to, since as it stands the source of the redundancy is that the two neurons in the first hidden layer would output identical values.

As for other benefits, numerical stability maybe?

I don't think that this is the intended solution haha, but I don't think You can get rid of the hidden layer all together due to the ReLU as mentioned above, so ¯_(ツ)_/¯

Q4 a) i)

We're going to use linearity of expectation.

Let $I_i, i \in \{1 \dots |V|\}$ be the indicator variable for whether or not the word i appears in Mary's sampling.

Since draws are with replacement and the distribution is uniform, for a token *not* to appear, all n draws must have chosen another token.

The probability of not choosing token i in a particular draw is of course $(|V|-1)/|V|$. The probability of not choosing token i over n samples is then

$$\left(\frac{|V|-1}{|V|}\right)^n \tag{57}$$

However, We are interested in the opposite event - We want to know the probability of *not* not choosing token i, which is given by

$$1-\left(\frac{|V|-1}{|V|}\right)^n \tag{58}$$

By linearity of expectation and the fact that the expected value of an indicator is simply the probability of it being 1, We get

$$E\left(\sum_{i=1}^{|V|} = I_i\right) = \sum_{i=1}^{|V|} E(I_i) = \sum_{i=1}^{|V|} P(I_i = 1) = |V| \cdot \left(1-\left(\frac{|V|-1}{|V|}\right)^n\right) \tag{59}$$

Q4 a) ii)

Given n draws, what is the probability that all the words will appear in the sample?

All of my attempts have failed, and a whole lot of searching leads me to conclude that the probability is:

$$\frac{|V|!}{|V|^n} S_2(n-1, |V|-1) \tag{60}$$

Where $S_2(n-1, |V|-1)$ is the Stirling number of the second kind.

Q4 b) i)

Alright, so, let X be the number of samples before "work" and "hard" bigram appears.

Let ps (partial success) mean that the word "work" has been sampled. Then by law of total expectation We can say

$$E(X) = (1 + E(X|ps)) \cdot P(ps) + (1 + E(X|ps')) \cdot P(ps') \tag{61}$$

Where We the 1's in there come from the fact that a single sample has been taken.

Now, given partial success We either fully succeed (fs) or fail and go back to the beginning, so:

$$E(X|ps) = (1 + E(X|ps, fs)) \cdot P(fs|ps) + (1 + E(X|ps, fs')) \cdot P(fs'|ps) \tag{62}$$
$$= (1) \cdot \frac{1}{|V|} + (1 + E(X)) \cdot \frac{|V| - 1}{|V|} \tag{63}$$

Where given full success, expected number of draws to get a success is zero. Given a failure We're back to $E(X)$.

Of course, given ps', the compliment of partial success i.e. partial failure, $E(X|ps') = E(X)$.

Putting it all together We get:

$$E(X) = (1 + E(X|ps)) \cdot P(ps) + (1 + E(X|ps')) \cdot P(ps') \tag{64}$$
$$= \left(1 + \frac{1}{|V|} + (1 + E(X)) \cdot \frac{|V| - 1}{|V|}\right) \cdot \frac{1}{|V|} + (1 + E(X)) \cdot \frac{|V| - 1}{|V|} \tag{65}$$
$$= \frac{1}{|V|} + \frac{1}{|V|^2} + (1 + E(X)) \cdot \frac{|V| - 1}{|V|^2} + (1 + E(X)) \cdot \frac{|V| - 1}{|V|} \tag{66}$$
$$= \frac{1}{|V|} + \frac{1}{|V|^2} + \frac{|V| - 1}{|V|^2} + \frac{(|V| - 1)E(X)}{|V|^2} + \frac{|V| - 1}{|V|} + \frac{E(X)(|V| - 1)}{|V|} \tag{67}$$

Now We just need to simplify this mess.

$$E(X) = \frac{1}{|V|} + \frac{1}{|V|^2} + \frac{|V| - 1}{|V|^2} + \frac{(|V| - 1)E(X)}{|V|^2} + \frac{|V| - 1}{|V|} + \frac{E(X)(|V| - 1)}{|V|} \tag{68}$$
$$E(X) - \frac{(|V| - 1)E(X)}{|V|^2} - \frac{E(X)(|V| - 1)}{|V|} = \frac{1}{|V|} + \frac{1}{|V|^2} + \frac{|V| - 1}{|V|^2} + \frac{|V| - 1}{|V|} \tag{69}$$
$$E(X)\left(1 - \frac{(|V| - 1)}{|V|^2} - \frac{(|V| - 1)}{|V|}\right) = \frac{1}{|V|} + \frac{1}{|V|^2} + \frac{|V| - 1}{|V|^2} + \frac{|V| - 1}{|V|} \tag{70}$$
$$E(X) = \frac{\frac{1}{|V|} + \frac{1}{|V|^2} + \frac{|V|-1}{|V|^2} + \frac{|V|-1}{|V|}}{1 - \frac{(|V|-1)}{|V|^2} - \frac{(|V|-1)}{|V|}} \tag{71}$$

Beautiful. If You plug $|V| = 2$ in there You get 6, which is the well known result of expected number of tosses to get 2 heads in a row when tossing a fair coin.

Q4 b) ii)

Well, the probability of "work" not appearing in n draws is

$$\left(\frac{|V|-1}{|V|}\right)^n \tag{72}$$

And of course the probability of it appearing is

$$1-\left(\frac{|V|-1}{|V|}\right)^n \tag{73}$$

And Mary wants this probability to be above 0.95:

$$1-\left(\frac{|V|-1}{|V|}\right)^n \geq 0.95 \tag{74}$$

$$-\left(\frac{|V|-1}{|V|}\right)^n \geq -0.05 \tag{75}$$

$$\left(\frac{|V|-1}{|V|}\right)^n \leq 0.05 \tag{76}$$

$$\log\left(\left(\frac{|V|-1}{|V|}\right)^n\right) \leq \log(0.05) \tag{77}$$

$$n \cdot \log\left(\frac{|V|-1}{|V|}\right) \leq \log(0.05) \tag{78}$$

$$n \geq \frac{\log(0.05)}{\log\left(\frac{|V|-1}{|V|}\right)} \tag{79}$$

$$\tag{80}$$

Where the last sign flips since log of a quantity less than 1 is negative.

Q4 c) i)

The answer here is identical to the answer to question **b)i)**, by the fact that the tokens are all equally likely.

Q4 c) ii)

Okay so We have

$$h_t^0 = f(w_0 x_{t-1} + w_1 x_t + w_2 h_{t-1}^0 + b_0) \tag{81}$$

So, that's our first hidden state. f is some arbitrary non-linearity (or maybe linearity, whatever, a function).

w_0 multiplies previous input, w_1 multiplies current input, w_2 multiplies the previous hidden state and We have a bias term.

The output then is

$$y_t = g(w_3 h_t^0 + b_1) \tag{82}$$

So w_3 is multiplying the hidden state, there's a bias term and another function.

The goal is to output x_{t-1} and x_t are the same word.

Okay so assume that the two words are equal and the bias is 0:

$$\begin{aligned} h_t^0 &= f(w_0 x_{t-1} + w_1 x_t + w_2 h_{t-1}^0 + b_0) && (83)\\ &= f(w_0 x_{t-1} + w_1 x_{t-1}) && (84)\\ &= f(x_{t-1}(w_0 + w_1)) && (85) \end{aligned}$$

So clearly one way to go is to let w_0 be -1 and w_1 be 1, so that when they are added, We get zero in the case of equality. Then f can just be like a check as to whether the input is 0, and if it is output 1, otherwise output 0.

g is then simply the identity function and b_1 is also 0.

So, $w_0 = -1$, $w_1 = 1$, $w_2 = 0$, $w_3 = 1$, all biases are zero, f is a boolean check for whether the input is 0 and g is the identity function.

Q4 c) iii)

So We simply use the second hidden layer as a counter. $w_4, w_5 = 1$, $b_1, b_2 = 0$ and g and h are still the identity function, and f is the boolean 0 check.

Q4 c) iv)

I think the non-uniform unigram language model has the greater probability of drawing two of the same token in a row.

This isn't a proof, but the reasoning is as follows: let p_i be the probability of drawing the i'th token. Now suppose We alter the distribution such that $p_i' = p_i - x$. Now, We'll have to add that x to some other p_j to preserve a distribution, and since the sample are identical, sampling p_j twice in a row will be more likely than if We had just left the probabilities alone.

Q5 a) i)
Well first let's re-derive this mess.

$$\sum_{\mathbf{t}\in\mathcal{T}^N} \exp\left\{\sum_{n=1}^{N} \text{score}\big(\langle t_{n-1}, t_n\rangle, \mathbf{w}\big)\right\} \tag{86}$$

So let's start with inside the bracket comes from

$$\text{score}\big(\mathbf{t}, \mathbf{w}\big) = \sum_{n=1}^{N} \text{score}\big(\langle t_{n-1}, t_n\rangle, \mathbf{w}\big) \tag{87}$$

So this is our simplifying assumption, and I suppose it's also what makes this a "Conditional Random Field". **t** stands for "tag" and it's the, well, tagging of the word sequence **w**. So We're saying that the score for the tag does not have to be calculated all in one go, but rather can be done in parts, where each part only depends on it's predecessor. Aight. Markov yay.

N is the length of the sentence by the way, so the length of both **t** and **w**.

$\mathcal{T}$ is the set of all possible tags for any particular word. Since there are N words, the total number of possible tags for **w** is then $\mathcal{T}^N$.

So then the expression is simply calculating the normalizing constant, since it's summing over all possible tags $\mathbf{t} \in \mathcal{T}^N$.

So, up next We get

$$\sum_{\mathbf{t}\in\mathcal{T}^N} \exp\left\{\sum_{n=1}^{N} \text{score}\big(\langle t_{n-1}, t_n\rangle, \mathbf{w}\big)\right\} \tag{88}$$

$$= \sum_{\mathbf{t}_{1:n}\in\mathcal{T}^N} \prod_{n=1}^{N} \exp\left\{\text{score}(\langle \mathbf{t}_{n-1}, \mathbf{t}_n\rangle, \mathbf{w})\right\} \tag{89}$$

Okay so what have We done here. For one, **t** now has a subscript. Sure.

We've converted the inner sum to a product, which is fine - before We were taking the exponent of a sum, but product of a bunch of exponents will sum the exponents so no worries there. We then still sum over every possible tag. Sure.

$$\sum_{\mathbf{t}\in\mathcal{T}^N} \exp\left\{\sum_{n=1}^{N} \text{score}\big(\langle t_{n-1}, t_n\rangle, \mathbf{w}\big)\right\} \tag{90}$$

$$= \sum_{\mathbf{t}_{1:n}\in\mathcal{T}^N} \prod_{n=1}^{N} \exp\left\{\text{score}(\langle \mathbf{t}_{n-1}, \mathbf{t}_n\rangle, \mathbf{w})\right\} \tag{91}$$

$$= \sum_{\mathbf{t}_{1:N-1}\in\mathcal{T}^{N-1}} \sum_{t_N\in\mathcal{T}} \prod_{n=1}^{N} \exp\left\{\text{score}(\langle \mathbf{t}_{n-1}, \mathbf{t}_n\rangle, \mathbf{w})\right\} \tag{92}$$

So, the first change is that the subscript for $\mathbf{t}$ now goes to $N-1$. So We're just taking out the last tag in the sequence of tags $\mathbf{t}_N$.

Since that last tag could be anything, We sum over all the possible tags, so $t_N \in \mathcal{T}$. There are $|\mathcal{T}|$ choices.

I mean really You could rewrite the first sum as N sums each over $|\mathcal{T}|$ terms each. So We're just splitting one out. No worries.

$$\sum_{\mathbf{t}\in\mathcal{T}^N} \exp\left\{\sum_{n=1}^{N} \text{score}\big(\langle t_{n-1}, t_n\rangle, \mathbf{w}\big)\right\} \tag{93}$$

$$= \sum_{\mathbf{t}_{1:n}\in\mathcal{T}^N} \prod_{n=1}^{N} \exp\left\{\text{score}(\langle \mathbf{t}_{n-1}, \mathbf{t}_n\rangle, \mathbf{w})\right\} \tag{94}$$

$$= \sum_{\mathbf{t}_{1:N-1}\in\mathcal{T}^{N-1}} \sum_{t_N\in\mathcal{T}} \prod_{n=1}^{N} \exp\left\{\text{score}(\langle \mathbf{t}_{n-1}, \mathbf{t}_n\rangle, \mathbf{w})\right\} \tag{95}$$

$$= \sum_{\mathbf{t}_{1:N-1}\in\mathcal{T}^{N-1}} \prod_{n=1}^{N-1} \exp\left\{\text{score}(\langle \mathbf{t}_{n-1}, \mathbf{t}_n\rangle, \mathbf{w})\right\} \times \sum_{t_N\in\mathcal{T}} \exp\left\{\text{score}(\langle \mathbf{t}_{N-1}, \mathbf{t}_N\rangle, \mathbf{w})\right\} \tag{96}$$

Okay so if I had to explain this in words, how would I do it?

Well, everything hinges on two parts - the first being that the score can be decomposed into partial computations. If there is no substructure, no simplification is possible.

The second I suppose is the distributative property I suppose. The key bit is that You can take stuff out of the product and sum it, and if You take the product of the stuff and the sum it all still works. Or rather, You have some operator B. B is iterating over all possible versions of the tags. Then You also have some operator A, and You can take stuff out of B, combine it with A and take output of A and stick it back into B and it all still works. Now You're reducing the exponential number of things You had to deal with. Sure.

Another way of saying it is that A has to give You the ability to group and it has to work with B via distributativity.

The viterbi algorithm just keeps track of max possible value for each node going backwards using multiplication and max.

Vertices V are the tags, edges E are pairs of tags, and the weight on each tag would be score(a, b). These should really be directed in our case but it ought to work regardless.

```
Data: Graph, source node
Result: Shortest path from source to all nodes
prq = initialize priority queue with source node with distance 0;
//scores is a dictionary of tuples such that
//scores[a] = (b, c) where c is the cumulative score of a
//and c is the node We took to arrive at a with score c
scores = empty dictionary;
//initialize scores to include source node scores[source] = (source, 1);
while prq not empty do
    //popMin retrieves and removes minimum entry in prq
    //the minimum is decided by the associated score
    //see prq.add below
    currentNode = prq.popMax() ;
    for neighbour in neighbours(currentNode) do
        //scores[neighbour](1) accesses the tuple for
        //neighbour and (1) accesses the cumulative score
        tempScore = scores[neighbour](1) + score(neighbour,
         currentNode)
        if tempScore > score[neighbour] then
            scores[neighbour] = (currentNode, tempScore)
            //tempScore is what is used to rank the options
            //and prq returns the neighbour entry
            //when popMax is called
            //loops will not enter prq due to increasing score
            //and anyway the graph We care about is acyclic
            prq.add(neighbour, tempScore)
        end
    end
end
```

Algorithm 1: Dijsktra's algorithm using a priority queue.

To get the best path, one simply picks the node they would like to find the path from, find the node in the scores list, look at it's predecessor, then look at the predecessor of the predecessor and so on.

Q5 a) ii)

Well the algorithm will visit every node once, so it's $\mathcal{O}(|V|)$ for that.

It'll also examine every edge twice - let a, b, be nodes, then it will look at the edge (a, b) first and sometime later at the edge (b, a), so $2|E|$ there or $\mathcal{O}(|E|)$. In those cases it will also consult the priority queue *prq*, and the complexity of that in the best case[1] is $\log(|V|)$, so in total We get $\mathcal{O}(|E| \cdot \log(|V|) + |V|)$.

In contrast, Viterbi visits each edge and node once, so We have $\mathcal{O}(|V| + |E|)$ time complexity. This obviously compares favourably with Dijsktra.

Now, since Dijsktra explores more promising venues first, it is possible to construct examples where, despite being worse asymptotically, Dijsktra will beat Viterbi.

[1] With Fibonacci heaps.

Likewise one could construct examples where Viterbi will beat Dijsktra. Neither is strictly better than the other, but Viterbi is better on average.

Q5 b) i)

Well first let's identify the semiring.

A are strictly positive real numbers of course.

$\bigoplus$ is the max operation.

$\bigotimes$ is the $\times$ operation, i.e. just simple multiplication. This works due to the fact that We're exponentiating, so taking the product leads to the sum in the exponent We want.

$\hat{0}, \hat{1}$ are 0 and 1 respectively.

In semiring notation We then get an algorithm that looks basically nothing like the first version:

```
Data: Graph $G$, score function, source node s
Result: Shortest path from source to all nodes
prq = initialize max queue with source node and score of 1;
scores = dictionary initialized with source node and value of 1;
while Q not empty do
    currentNode = prq.popMax();
    for each neighbour of currentNode do
        tempScore = score[neighbour] $\bigoplus$ (scores(currentNode) $\bigotimes$
         score(neighbour) );
        if scores[neighbour](1) $<$ tempScore then
            //scores[neighbour]= parent, new best score
            scores[neighbour] =(currentNode, tempScore);
            //priority decided based on score obviously
            prq.add(neighbour, tempScore);
        end
    end
end
```

Algorithm 2: Dijsktra's algorithm using a priority queue in semiring notation.

It's a little different since I don't like initializing the queue with the entire graph at the beginning. Instead, the queue is initialized with just the source and entries are added as You go. This could be dicey in cyclic settings, but We don't have to worry about cycles in our case. Don't think I'd use this with a cyclic graph, since cycles would just continue increasing the score and get continuously added.

To get the path one traverses the parents in scores, same as before - each node has a parent, and to get the path You take the parent of the goal node, then the parent of the parent and so on.

Q5 b) ii)

I think just using $\bigoplus$ = min operation should work, and of course the priority queue would now use popMin instead of popMax, and also the if condition changes to $>$. I think this would not work in general due to cycles, but our graph is acyclic.

Q5 b) iii)

Well, I think the $\bigotimes$ = min, since given a path We'll want to take the minimum out of all the edge values.

Could You use $\bigoplus$ = max? I think that's it. Across an entire path We want to take the minimum to find the minimal value, but when choosing amongst a number of possible edges to take, We'll choose the maximum.