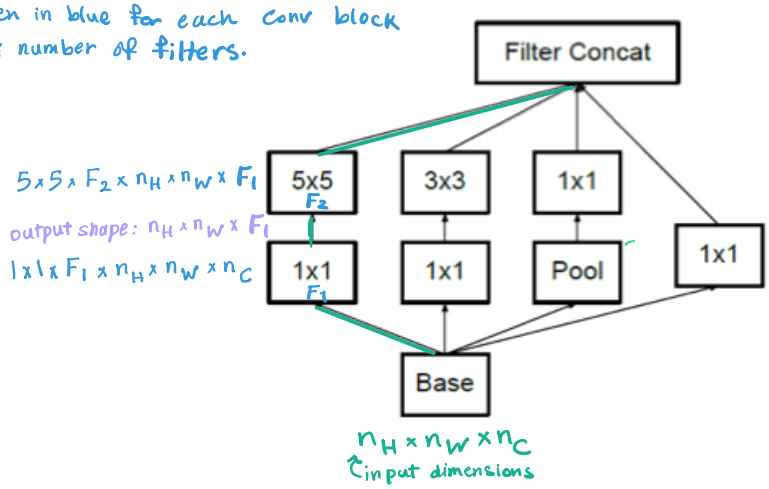
ML_HW4_ Lachin Naghashyar 98110179

1 Modern CNN

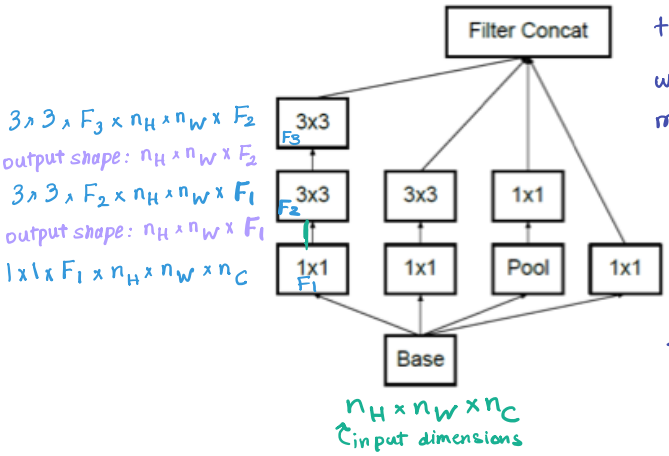
You have been introduced to the original inception module.

Conv ops are written in blue for each conv block and each have F_i number of filters.



1.1 (5 Points)

Show that the following module is more parameter efficient than the original one.



total number of computations in the previc
which corresponds to the version 1 of
module (with dimension reduction):

$n_H\, n_W\, F_1\,(n_C + 25\,F_2)$

Number of computations here:

$n_H\, n_W\,(F_1 n_C + 9\,F_1\,F_2 + 9\,F_2 F_3)$

Second − First $= n_H\, n_W(\ 9\,F_2(F_1+F_3) - 2$

$= n_H\, n_W(\ -14\,F_2 F_1 + 9\,F_2 F$

$= n_H n_W\,(-F_2(14F_1 - 9\,F_3)$

if $F_1=F_2=F_3=1 \rightarrow$ Second − First $= n_H n_W\,(-7)$

Factorizing 5x5 convolution into two 3x3 convolutions is one of the improvements in the second version of the Inception module. This improves the computational speed. Using a 5x5 convolution is 2.78 times more expensive than a 3x3 convolution. Hence, stacking two 3x3 convolutions, boosts the performance.

Moreover, in terms of number of parameters, using a 5x5 convolution requires 5x5=25 weight parameters. However, using two 3x3 filters, requires 2x3x3 = 18 parameters which is less. When we have param numbers decrease from 25 to 18, we see a reduction in computatio by 28%.

1.2 (5 Points)

Show that you can save more computation by factorizing 3×3 convolutions into two 2×2 convolutions. (One branch is enough.)

Assume that we have an input of size $x \times y$ and we want to have an output of size $x \times y$

Now lets compare the number of operations between using a 3x3 convolution vs. two stacked 2x2 convolutions.

- using one 3x3 block: $3 \times 3 \times x \times y$
- using two 2x2 blocks: $2 \times (2 \times 2) \times x \times y$

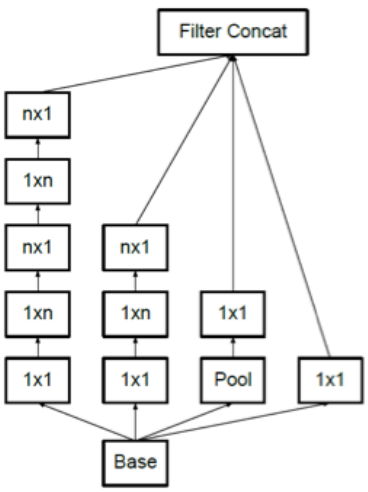
→ computation cost is reduced by a factor of:

$$\frac{2 \times 2 \times 2 \times x \times y}{3 \times 3 \times x \times y} = \frac{8}{9} \sim 88\%$$

This results in a computation saving of around 11 percent.

1.3 (10 Points)

Show that using asymmetric convolutions as follows saves computation. (One branch is enough.) How do you compare this enhancement with the previous one?



Using a nxn convolution is same as using a nx1 convolution following a 1xn convolution.

For example, assume that we use a combination of 1x3 and 3x1 convolutions instead of a 3x3 one.

Again consider the shapes of input & output to be $x \times y$ then we can calculate the number of computations as follows:

- a single 3x3: $x \times y \times 3 \times 3$
- A 1x3 and a 3x1: $1 \times 3 \times x \times y + 3 \times 1 \times x \times y = 2 \times x \times y \times 3 \times 1$

→ $\frac{2}{3}$ → we have computation saving of 33%.

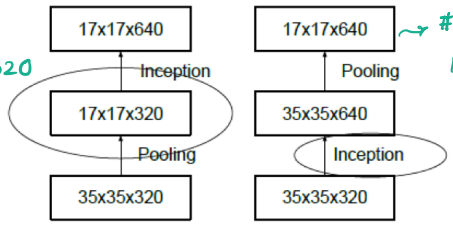
Compared to the previous part, it saves less computational load

Moreover, some kernels are spatially separable convolution. Meaning if a kenel is 3x3 forexample, then convolving it is equivalent to convolving on 1x3 and then on 3x1. This would require 6 parameters instead of 9 while doing the same operation.

In the more general case, we need to update $n+n=2n$ parameters instead of n^2 → we have computatio saving of $1-\frac{2}{n}$

1.4 (10 Points)

Consider the following methods for grid size reduction. Which one do you prefer in terms of computational cost? Which one creates a representational bottleneck?



$17\times17\times320 + 17\times17\times640\times K^2\times320$

#computations = $17\times17\times640 + 35\times35\times640\times K^2\times320$

$$\frac{17\times17\times640 + 35\times35\times640\times K^2\times320}{17\times17\times320 + 17\times17\times640\times K^2\times320} = \frac{2+8t}{1+2t} \sim 4$$

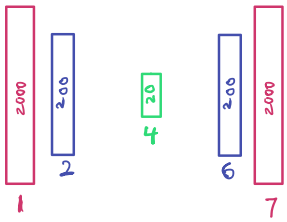
The right one first expands the filter bank and then reduces its size. However, in the left one we first reduce the grid size and then expands which itroduces a representational bottleneck. The right one is more expensive, however, it makes more sense to expand first and then shrink (get the general idea and then specialize) rather than first reducing the size and then expanding. which is what happening in the Left one

2 Autoencoders

1000 gene expression profile samples are given to us. These samples contain normal and cancer cases. The objective is to design a model which can distinguish cancer samples from normal ones. To do so, we have split the data into 800 train and 200 test samples. Each gene expression profile is a 20,000 dimension vector of numbers between 0 and 15. We have designed a 7-layer Autoencoder. The first and last layers have 2000 neurons, the second and 6th layers have 200 neurons and the fourth layer has only 20 neurons. Consider that all of the layers are fully-connected layers and the activation function for all of the neurons is sigmoid. The loss function has been chosen to be MSE.

2.1 (5 Points)

We have trained the network for 100 epochs but the loss function value does not decrease. What is the main reason?



Some solutions:

- Since the activation function is sigmoid, normalizing data helps with the convergence speed.
- Sigmoid can also cause vanishing gradients which should be prevented.
- A very high learning rate may get you stuck in an optimization loop or get you too far from any local minima.
- Most blogs (like Keras) use binary crossentropy as their loss function, but MSE is not "wrong"
- Having loss staying steady in a certain amount might be because the model is predicting the same value for every input and this might be a reason of setting all weight to zero.
- Try different weight initializations, activation functions, loss function, optimizer,...

Below are some things we can do to help decrease the loss in AEs:

- Reduce mini_batch size: Having smaller batch size will make the gradient more noisy. when it's backpropagating. This noise in gradient descent could help the descent overcome local minimas.
- Shift numbers to -7 to 8
- We are trying to lower our loss, but to what end? Because as our latent dimension shrinks, the loss will increase but the autoencoder will be able to capture the latent information of the data better. (Since you're forcing the encoder to represent an info of higher dimension with an information in lower dimension)

2.2 (10 Points)

After solving the problem, we again train the network for 100 epochs. We observe that MSE is less than 0.1 for test samples. Can we thereby conclude that the network is performing well? If not, what should we check?

Yes we can concolude that the network is performing well if the loss of 0.1 is not large compared to the magnitude of our data. However, if our data is not in that order of magnitude, then we can concolude that the network is not performing well. Hence, keeping this in mind, it's probably acceptable since the ideal MSE isn't 0 (then your model is overfitting on your train data). We can also use other metrics and compare the results to be sure.

2.3 (10 Points)

After the training process, we notice that with every small change in input values, a lot of neurons from the latent layer will change. What should we do in order to solve this issue? Provide a formula if possible.

A contractive autoencoder learns representations that are robust to a slight variation of the input data. The idea behind it is to map a finite distribution of input to a smaller distribution of output. This idea essentially trains an autoencoder to learn representation even if the neighboring points change slightly.

It uses the formula $\mathcal{L}(x, x') + \lambda \sum_i \| \nabla_x h_i \|^2$.
The first part is a penalizing term as before. In the second part, it uses the squared Frobenius norm of the Jacobian matrix J. The term describes the gradient field of the latent representation h with respect to input x. This term penalizes the large derivatives of the jacobian matrix or gradient field of the latent representation h. Any small changes in the input data that leads to large change or variation in the representational space is penalized.

We can also make use of VAEs. which instead of encoding the input into a single point, we encode it into a distribution over the latent space. We will be encoding into, and sampling from normal dists. So encoder needs to return the mean & variance matrix, describing these Gaussians.

$$\text{loss} = \underbrace{\|x - \hat{x}\|^2}_{\text{reconstruction term}} + \underbrace{KL[N(\mu_x, \sigma_x), N(0,1)]}_{\text{regularization term}}$$

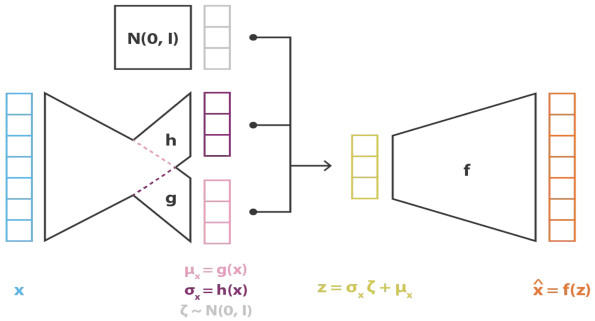
2.4 (10 Points)

Finally we succeed to train the network in a way that with low test error we can reconstruct input samples from the 20 latent neurons. What property of our input data has helped us in this success?

fixing the distribution that our data comes from can help us in VAE.

In practice, we use two networks at the end of encoder to compute the mean & variance.

Problem: We can't perform BP on the direct sampling operation → Solution: we sample from a standard normal dist



4 Sequence-to-Sequence Models Comprehension

Answer the following questions

4.1 (15 Points)

How can vanishing and exploding gradients be controlled in RNN?

very very deep NNs are difficult to train because of exploding & vanishing gradients problems. This is done by adding a short-cut or skip connection. and instead of having $a^{[l+1]} = g(z^{[l+2]})$, we will have: $a^{[l+1]} = g(z^{[l+2]} + a^{[l]})$ This way, if z_i s become really small and close to zero, adding $a^{[l]}$ term prevents the vanishing gradients problem. Hence, we can overcome the optimization problem caused in deeper models.

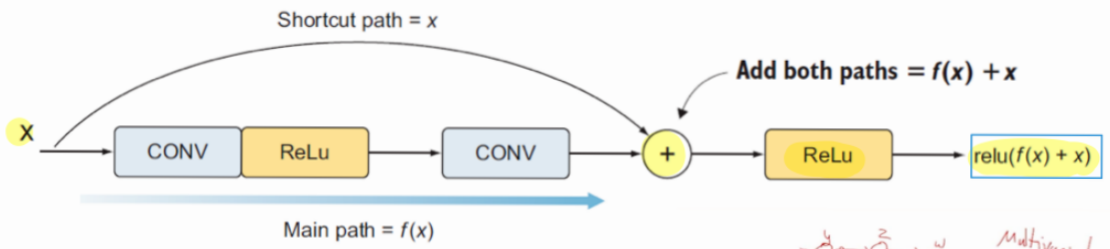
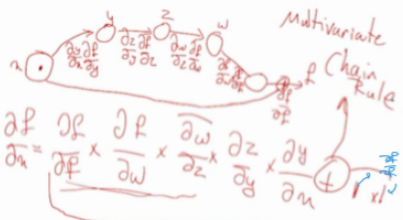


Figure: [1]



Other solutions → Vanishing: Use some other activation function such as leaky relu instead of relu. Or use some other weight initialization (for example not constan)
↳ Exploding: Gradient clipping by norm

4.2 (5 Points)

What are the advantages of bi-directional LSTM over simple LSTM?

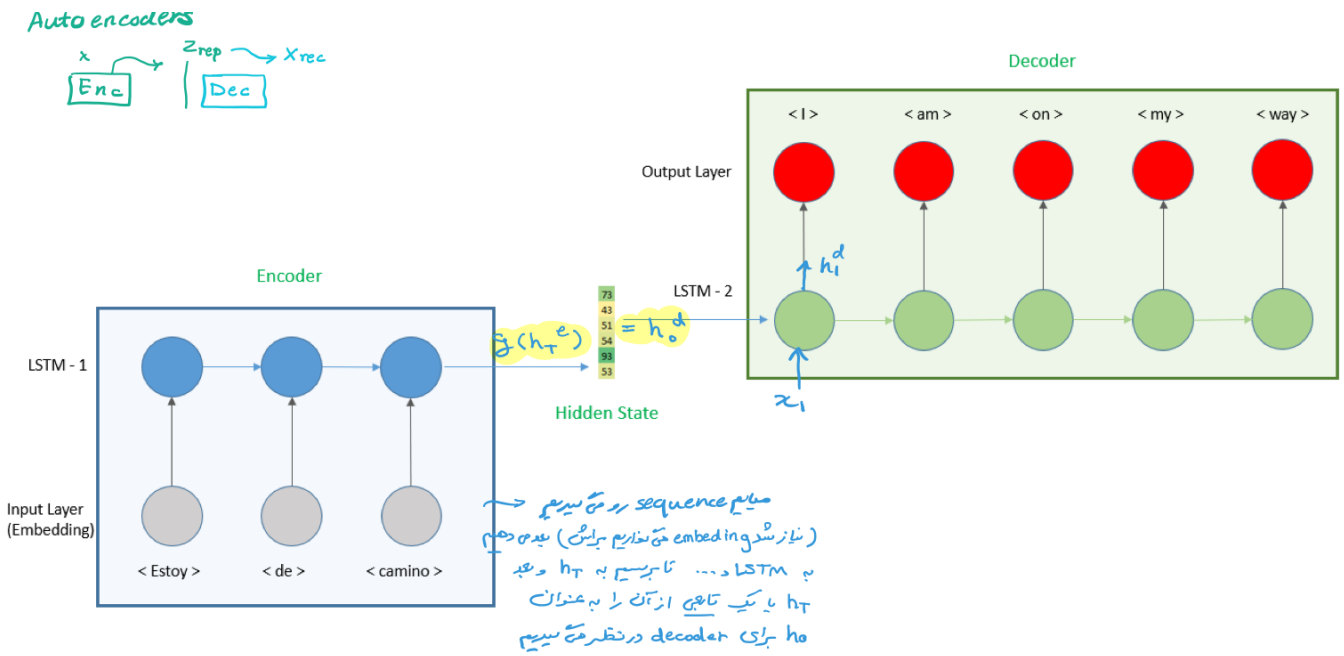
In the standard LSTM we can make input flow in one direction, either backward or forward. However, in bidirectional LSTM, we can make input flow in both directions and it's capable of utilizing information in both directions to preserve the future and the past information.

Moreover, bidirectional LSTM models offer better predictions than regular LSTM based models and they spend more time during training which may hinder its applicability on large datasets. It also has much better speed at which we get the desired results and drastically reduces the time taken by the search by having simultaneous searches. It also saves resources for users as it requires less memory capacity to store all the searches.

4.3 (10 Points)

Why is encoder-decoder architecture used in Machine Translation? Explain problems of this architecture.

If we want an output with different length from input, Encoder-Decoder architecture could be a solution. The approach involves two RNNs, one to encode the input sequence, called the encoder, and a second one to decode the encoded input sequence into the target sequence called the decoder.



In fact, when we use an encoder, the meaning of the sentence will be stored and represented by a vector. We can't directly convert a sentece in French into English because we would lose the context. If we translate directly, we would translate word by word, without caring about the global meaning of the sentence.

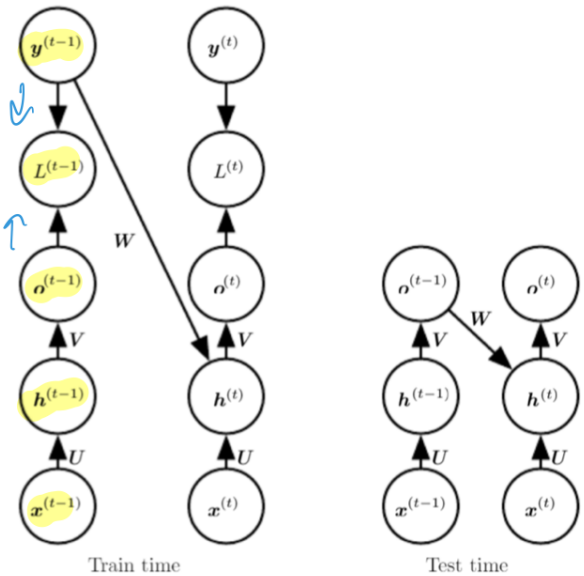
Using Encoder_Decoder architecture, we can obtain a vector (h_T^e) in the Encoder, which helps us keep the meaning of the sentence.

One of the limitations of this architecture is that by increasing the length of the input sequence, the model captures the essential information roughly and may preserve the information from initial blocks much less (vanishing gradients problem).

Similarly, if the output seq is too long, then again we will have the vanishing gradients problem.

We can use Teacher Forcing to train the RNNs more efficiently. It works by using the actual output at the current time step $y^{(t-1)}$ as input in the next time step, rather than the $o^{(t-1)}$ generated by the network.

هر اشتباه رو کپ و در همونجا جریمه می‌کنه
فقط (توی قبلی این خطاها انباشته
می‌شد) و هم چنین درستش رو بهش
می‌دیم.



4.4 (10 Points)

What is beam search algorithm? Explain this algorithm briefly.

In the search strategy for prediction, we want to find $\tilde{y}_T, \tilde{y}_{T-1}, \ldots, \tilde{y}_1$ in such way that they maximize the term $P(\tilde{y}_T, \tilde{y}_{T-1}, \ldots, \tilde{y}_1 | S)$. We could do this by exhaustive search over all possible combinations of $\tilde{y}_T, \ldots, \tilde{y}_1$. However, this strategy is costly and of $O(V^T)$.

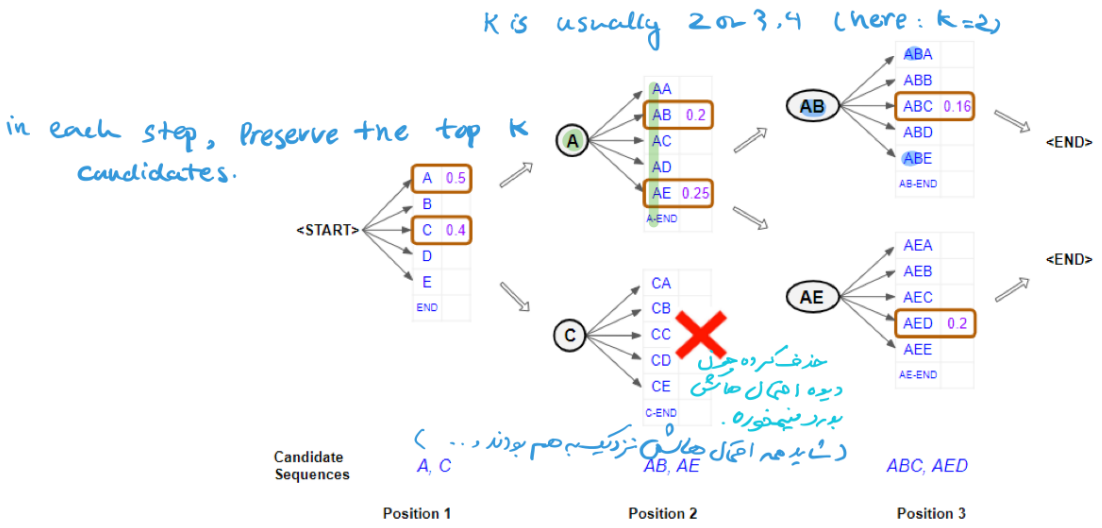
Moreover, another approach can inlove a greedy search in which we choose a state that maximizes the probability in that time step given the previous choices. Although it costs much less (O(TV)), it can be very inaccurate sometimes.

$$max\, P(y_T, y_{T-1}, \ldots, y_1|S) \approx \prod_{t=1}^{T} max\, P(y_t|y_{t-1}, y_{t-2}, \ldots, y_1, S)$$

P(*|A) P(*|A,B) P(*|A,B,C)

Time step	1	2	3	4
A	0.5	0.1	0.2	0.0
B	0.2	0.4	0.2	0.2
C	0.2	0.3	0.4	0.2
<eos>	0.1	0.2	0.2	0.6

Beam search however, is something between two previous methods. It's a heuristic search technique that always expands the k number of the best nodes at each level. It progresses level by level and moves downwards only from the best K nodes at each level of the tree. However, at each level, it only evaluates a K number of states. Other nodes are not taken into account.

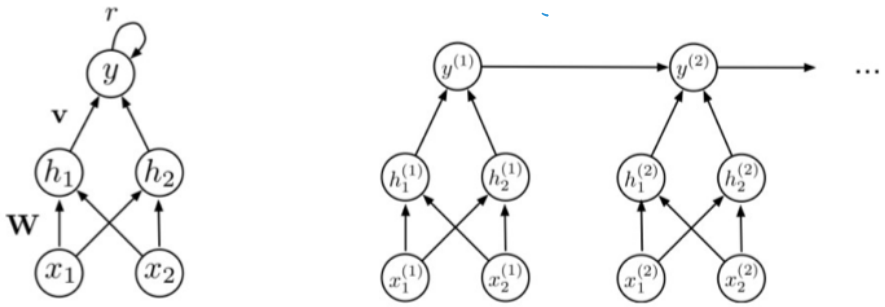


It has a time complexity of O(log(K) K VT) which is much better than the exhaustive search. It also performs much better than the greedy approach.

5 RNN Calculation

5.1 (20 points)

We want to process two binary input sequences with 0-1 entries and determine if they are equal. For notation, let $x_1 = x_1^{(1)}, x_1^{(2)}, \ldots, x_1^{(T)}$ be the first input sequence and $x_2 = x_2^{(1)}, x_2^{(2)}, \ldots, x_2^{(T)}$ be the second. We use the RNN architecture shown in the Figure.



We have the following equations in which W is a 2×2 matrix, b is a two dimensional bias vector, v is a two dimensional weight vector and r, C, C_0 are scalars.

$$h^{(t)} = g(Wx^{(t)} + b)$$

$$y^{(t)} = \begin{cases} g(v^T h^{(t)} + r y^{(t-1)} + C) & \text{if t} > 1 \\ g(v^T h^{(t)} + C_0)) & \text{if t} = 1 \end{cases}$$

$$g(z) = \begin{cases} 1 & \text{if z} > 1 \\ 0 & \text{if z} \leq 1 \end{cases}$$

Find W, b, v, r, C, C_0 such that at each step $y^{(t)}$ indicates whether all inputs have matched up to the current time or not.

we can have $h_1^{(t)}$ determining if both inputs are 0 and $h_2^{(t)}$ detemine if both inputs are one.

$$h^{(t)} = g(W x^{(t)} + b) \quad : \quad \begin{bmatrix} h_1^{(t)} \\ h_2^{(t)} \end{bmatrix} = g\left(\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \begin{bmatrix} x_1^{(t)} \\ x_2^{(t)} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right) = \begin{bmatrix} g(W_{11} x_1^{(t)} + W_{12} x_2^{(t)} + b_1) \\ g(W_{21} x_1^{(t)} + W_{22} x_2^{(t)} + b_2) \end{bmatrix}$$

$y^{(t)}$ → if t=1 :

$$y^1 = g\left(\begin{bmatrix} v_1 & v_2 \end{bmatrix} \begin{bmatrix} h_1^{(t)} \\ h_2^{(t)} \end{bmatrix} + C_0 \right) = g(v_1 h_1^1 + v_2 h_2^1 + C_0)$$

we should have: $(h_1^1, h_2^1) = (0,1) : v_2 + C_0 > 1$ we can have $C_0 = 0.5$ and $v_1 = v_2 = 1$

$(h_1^1, h_2^1) = (1,0) : v_1 + C_0 > 1$

$(h_1^1, h_2^1) = (0,0) : C_0 \leq 1$

Also we should have: $(x_1^{(t)}, x_2^{(t)}) = (0,0) \rightarrow b_1 > 1$ we can have: $b_1 = 1.5$ and $W_{11} = W_{12} = -1$

$(x_1^{(t)}, x_2^{(t)}) = (0,1) \rightarrow W_{12} + b_1 < 1$

$(x_1^{(t)}, x_2^{(t)}) = (1,0) \rightarrow W_{11} + b_1 < 1$

$(x_1^{(t)}, x_2^{(t)}) = (1,1) \rightarrow W_{12} + W_{11} + b_1 < 1$

Similarly: $(x_1^{(t)}, x_2^{(t)}) = (1,1) \rightarrow W_{21} + W_{22} + b_2 > 1$

$(x_1^{(t)}, x_2^{(t)}) = (0,1) \rightarrow W_{22} + b_2 < 1$

$(x_1^{(t)}, x_2^{(t)}) = (1,0) \rightarrow W_{21} + b_2 < 1$

$(x_1^{(t)}, x_2^{(t)}) = (0,0) \rightarrow b_2 < 1$ we can have: $b_1 = -0.5$ and $W_{11} = W_{12} = 1$

also for the case of $t > 1$ for $y^{(t)}$ we have:

$$y^{(t)} = g\left(\begin{bmatrix} v_1 & v_2 \end{bmatrix} \begin{bmatrix} h_1^{(t)} \\ h_2^{(t)} \end{bmatrix} + r\, y^{(t-1)} + C \right) = g\left(v_1 h_1^{(t)} + v_2 h_2^{(t)} + r y^{(t-1)} + C\right)$$

we should have:

$(y^{(t-1)}, h_1^{(t)}, h_2^{(t)}) = (1,0,0) \rightarrow r + C < 1$

$(y^{(t-1)}, h_1^{(t)}, h_2^{(t)}) = (1,0,1) \rightarrow 1 + r + C > 1$

$(y^{(t-1)}, h_1^{(t)}, h_2^{(t)}) = (1,1,0) \rightarrow 1 + r + C > 1$ we can set $C = -0.5$ and $r = 1$

$(y^{(t-1)}, h_1^{(t)}, h_2^{(t)}) = (0,0,0) \rightarrow C < 1$

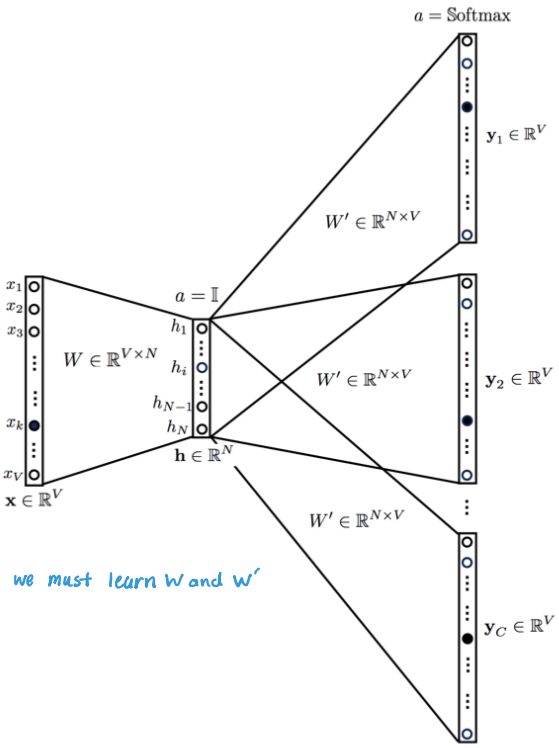
$(y^{(t-1)}, h_1^{(t)}, h_2^{(t)}) = (0,1,0) \rightarrow 1 + C < 1$

$(y^{(t-1)}, h_1^{(t)}, h_2^{(t)}) = (0,0,1) \rightarrow 1 + C < 1$

6 Word Embedding

6.1 (20 points)

By using the following figure(next page), explain skip-gram. Your explanation must include the probability the model wants to estimate, the concept of the context window, the output of the model, and the loss function used in the training.



we must learn W and W'

Using language models, we can assign a probability to a sequence of tokens and this way, a good model will assign high probability to senteces which are valid both syntactically and semantically. (For example, using a Bigram model, we can let the probability of the seq depend on the pairwise probability of a word in the seq and the word next to it: $P(w_1, \dots, w_n) = \prod_{i=2}^{n} P(w_i | w_{i-1})$

Another approach is to creat a model such that given the center word, it can predict or generate its surrounding words(context words in which we can have C to be the number of words surrounding a center word. For instance, $C=2$ context of the word "fox" in the sentence "The quick brown fox jumped over the lazy dog" is {"quick", "brown", "jumped", "over"}.)

In this model, our input is one hot vector (center word) which we represent with x. And the output vectors as $y^{(i)}$. We creat two matrices; $\mathcal{V} \in \mathbb{R}^{n \times |V|}$ and $\mathcal{U} \in \mathbb{R}^{|V| \times n}$. Where n defines the size of our embedding space and $|V|$ is the size of our vocabulary. $\mathcal{V}$ is the input word matrix such that the i-th column of $\mathcal{V}$ is the n-dimensional embedded vector for word w_i when it is an input to this model. Similarly, $\mathcal{U}$ is the output of word matrix. The j-th row of $\mathcal{U}$ is an n-dimensional embedded vector for word w_j when it is an output of the model. (Note that w_i is word i from vocabulary V and for every word w_i, we learn two vectors, the input word vector v_i and the output word vector u_i)

We can breakdown the way this model works in 6 steps:

1. Generate the onehot input vector x
2. Get the embedded word vectors for the context $v_c = \mathcal{V}x$
3. Set $\hat{v} = v_c$
4. Generate $2m$ score vectors, $u_{c-m}, \dots, u_{c-1}, u_{c+1}, \dots, u_{c+m}$ using $u = \mathcal{U}v_c$
5. Turn each of these scores into probabilities, $y = \text{softmax}(u)$
6. We desire our probability vector generated to match the true probabilities which is $y^{(c-m)}, \dots, y^{(c-1)}, y^{(c+1)}, \dots, y^{(c+m)}$, the onehot vectors of the actual output.

We need to generate an objective function to evaluate the model. We do this using a Naive Bayes independance assumption to break out the probabilities (given the center word, we assume all outputs are completely independant).

$$\text{minimize } J = -\log P(w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m} \mid w_c) = -\log \prod_{j=0, j\neq m}^{2m} P(w_{c-m+j} \mid w_c)$$

$$= -\log \prod_{j=0, j\neq m}^{2m} P(w_{c-m+j} \mid v_c) = -\log \prod_{j=0, j\neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{K=1}^{|V|} \exp(u_K^T v_c)}$$

$$= \sum_{j=0, j\neq m}^{2m} u_{c-m+j}^T v_c + 2m \log \sum_{K=1}^{|V|} \exp(u_K^T v_c)$$

Using this objetive function, we can compute the gradients with respect to the unknown parameters and at each iteration update them via SGD

6.2 (20 points)

Suppose that the context window (C) is equal to 1. (For example for each center word, we just consider its left neighbor) We also denote the loss function as

$$L(x, \hat{y}, W, W')$$

In which $\hat{y}$ is the one-hot encoded vector of the word in the context window and x is the one-hot encoded vector of the center word.
Using the previous part, calculate $\frac{\partial L}{\partial w_k}$ in which w_k is the kth row of matrix W. For simplification purposes, you can assume that $\frac{\partial Softmax(y)}{\partial y}$ is given as $S'(y)$. However, we highly recommend a complete calculation.

$C=1 \rightarrow h = W^T x$, $u = u_1 = W'^T W^T x$ and $y = y_1 = \text{softmax}(u)$

$\rightsquigarrow \mathcal{L}(u(W, W')) = \mathcal{L}(u_1(W, W'), \dots, u_{|V|}(W, W'))$

$$\frac{\partial \mathcal{L}}{\partial w'_{ij}} = \sum_{k=1}^{|V|} \frac{\partial L}{\partial u_k} \frac{\partial u_k}{\partial w'_{ij}} \quad , \quad \frac{\partial \mathcal{L}}{\partial w'_{ij}} = \sum_{k=1}^{|V|} \frac{\partial L}{\partial u_k} \frac{\partial u_k}{\partial w'_{ij}}$$

Note that weight w'_{ij}, connects a node i of the hidden layer to a node j of the output layer and it only affects the output score u_j. Hence, among all the derivatives, $\frac{\partial u_k}{\partial w'_{ij}}$, only one where $k=j$ will be different from zero $\rightarrow \frac{\partial \mathcal{L}}{\partial w'_{ij}} = \frac{\partial \mathcal{L}}{\partial u_j} \frac{\partial u_j}{\partial w'_{ij}}$

Note that $\frac{\partial \mathcal{L}}{\partial u_j} = -\underbrace{\delta_{jj^*}}_{\text{it's equal to one if } j=j^* \text{ o.w. } =0} + y_j := e_j$ ← this vector represents the difference between the target and the predicted output

$$\rightarrow \frac{\partial u_j}{\partial w'_{ij}} = \sum_{k=1}^{|V|} w_{ik} x_k \Rightarrow \frac{\partial \mathcal{L}}{\partial w'_{ij}} = (-\delta_{jj^*} + y_j) \left(\sum_{k=1}^{|V|} w_{ki} x_k \right)$$

Similarly we have: $\frac{\partial u_k}{\partial w_{ij}} = w'_{jk} x_i$ ← only when $t=i$ or $s=j$ (note that $u_k = \sum_{t=1}^{N} \sum_{s=1}^{N} w'_{tk} w_{st} x_s$)

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_{k=1}^{|V|} (-\delta_{kk^*} + y_k) w'_{jk} x_i$$