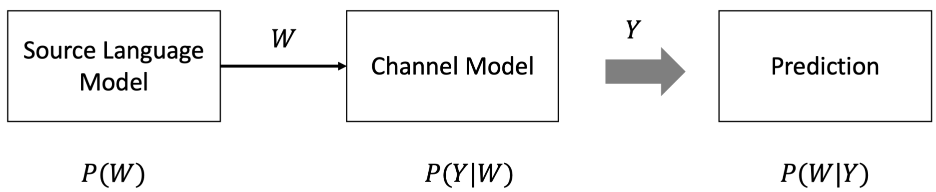
Language Modelling

Language modelling is the task of predicting the next word, given some context.

$$w^* = P(w|context)$$

1 Noisy Channel Model



The noisy channel model is derived from the language model as follows:
If the context is Y and word is W, we need to predict the next word, W*

$$\begin{aligned} W^* &= argmax_w \ P(W|Y) \\ &= argmax_w \ [\frac{P(Y|W).P(W)}{p(Y)}] \\ &\approx argmax_w \ P(Y|W).P(W) \\ &= argmax_w \ [channelModel * LanguageModel] \end{aligned}$$

1.1 Examples of noisy channel model

1. **Predictive keyboard**

   Predict the probability of a sentence (or some typed input) given keystrokes

$$\begin{aligned} W^* &= argmax_w \ P(W|K) \\ &\approx argmax_w \ P(K|W).P(W) \\ &= argmax_w \ [keystrokeModel * LanguageModel] \end{aligned}$$

2. Language Translation

Predict the probability of a sentence in English given input in French.

$$\begin{aligned} E^* &= argmax_w \; P(E|F) \\ &\approx argmax_w \; P(F|E).P(E) \\ &= argmax_w \; [TranslationModel * LanguageModel] \end{aligned}$$

3. Speech Recognition

Predict the probability of a sentence in English given input audio.

$$\begin{aligned} E^* &= argmax_w \; P(E|audio) \\ &\approx argmax_w \; P(audio|E).P(E) \\ &= argmax_w \; [TranslationModel * LanguageModel] \end{aligned}$$

4. Optical Character Recognition

Predict the probability of a word in English given input pixels.

$$\begin{aligned} E^* &= argmax_w \; P(E|pixels) \\ &\approx argmax_w \; P(pixels|E).P(E) \\ &= argmax_w \; [TranslationModel * LanguageModel] \end{aligned}$$

Other examples include spelling correction, handwriting recognition etc.

1.2 Probabilistic Language Model

Goal: Assign useful probabilities P(x) to words or sentences x.
Input: Many observations of words or sentences from available the training data.
Output: A system capable of computing P(x)

- P(x) would indicate the following:
  - Plausibility of x given the context in our training data.
    - i.e. probability of a word or sentence we compute would depend on the domain, context etc. of the data we use for training.
    - It doesn't check grammar

1.2.1 N-Gram Model

In our language model, p(x), how do we represent x?

$$p(x) = p(w_1, w_2, \dots, w_n)$$
$$= \prod_t p(w_t | w_{t-1}, w_{t-2}, \dots, w_0)$$

Now, this above value is hard to compute for the following reasons:

- As word histories grow larger, computing the probabilities get tougher.
- **Sparsity**: As word history grows, the probability of occurrence becomes very small. For example, the number of samples we have with a 10-word long sentence in our corpus could be extremely small or even 0. This would make our calculations inaccurate.

We use the **Markov assumptions** (even though we know it is not true for language) to overcome the sparsity problem. This gives us:

$$p(x) = p(w_1, w_2, \dots, w_n)$$
$$= \prod_t P(w_t)$$

(Uni-Gram Model, Bag of words assumption – All are independent)

$$= \prod_t P(w_t | w_{t-1})$$

(Bi-Gram Model, Markov assumption – Word depends only on previous word)

$$= \prod_t P(w_t | w_{t-1}, w_{t-2})$$

(Tri-Gram Model, Markov assumption – Word depends only on last two words)

1.2.1.1 Parameter estimation

How do we estimate the parameters of an N-Gram model?
- The maximum likelihood estimator (θ) is the relative frequency of the n-grams
- This gives us (Bi-Gram model):

$$\theta = P(w_t | w_{t-1}) = \frac{count(w_{t-1}, w_t)}{\sum_w count(w_{t-1}, w^*)}$$

Example:

$$P(door|the) = \frac{count\ of\ "the\ door"}{Total\ count\ of\ "the\ *"}$$

1.2.1.2 Higher order N-Grams

As the N-gram gets larger, the counts get smaller.
This makes their distribution sharper and our estimates worse.

1.2.1.3 Generative Process

Once we compute the language model (relative frequencies of the N-grams), we have a generative model. This works as follows:
- Pick an n-gram repeatedly until we pick a [stop] placeholder.

1.2.1.3.1 Unigram Model

- Each word in the vocab is considered independent
- Generative process: Pick a word repeatedly until we pick STOP.
- Problem:
  - Common words keep getting picked
  - P(the the the) >> p(I like ice cream)

1.2.1.3.2 Bi-Gram model

- Each word is conditioned on the previous word.
- **Generative process**: Start with a word. Then pick a word based on the previous word repeatedly until we pick STOP.
- Problems with N-Gram
  - As N-gram size increases, the generative process improves.
  - However, our vocab size increases.
  - Also, other problems with sparsity arise.
  - Not possible to model long range dependencies.

1.2.1.3.3 Character LMs

- Each character is conditioned on the previous character(s).
- Average adult English vocabulary size is about 30k words.
- Word-based LMs can get very large.
- What about LMs based on characters?
  - English vocabulary for characters is small.
  - Trade off vocabulary size for *n*-gram order.
  - Character LMs can capture morphology.
    - **Example**: Even if a word in the vocab doesn't end with "-ing", character LMs can generate words ending with "-ing" correctly.

1.2.2 Measuring Model Quality

- Increasing the order of N-grams gets us better sentences.
- However, with our MLE (relative frequencies), higher order gives us better likelihood only on training data, not the test data.
- So, how do we determine how good a model is?
- **Extrinsic (in-vivo) Evaluation**
    - We train parameters on a training set.
    - We test model performance on a test set of unseen data.
    - Depending on the task, we compare accuracy between various models.
    - For example:
        - Spelling corrector: how many misspelled words were corrected properly
        - Translation: How many words translated correctly
    - This is called extrinsic evaluation because we are looking at something external to our model (say n-gram) to evaluate model performance.
    - Problem with extrinsic evaluation
        - Time consuming

- **Intrinsic Evaluation**
    - Evaluate if the model is useful in and of itself.
    - This is about evaluating the language model itself and not about any particular application.
    - Approach like **perplexity** is a bad approximation of extrinsic evaluation.
        - Unless, training data looks a lot like the test data.
        - Generally useful in pilot experiments.

1.2.2.1 Perplexity (Intrinsic evaluation)

- A good language model is one that assigns the highest probability to the word that actually occurs.
- The best language model is one that best predicts unseen text
- For N-gram models, we can use entropy to determine this.
    - **Entropy** is defined as:

$$\boldsymbol{H(X) = -\sum_i P(x_i)\, log_2\, P(x_i)}$$

Where,

$P(x_i)$ is the probability of occurrence of x_i

Example for Bi-Gram Model we use a per-word entropy test.

$$H(X|\theta) = -\sum_{x\in X} P(x_i|x_{i-1},\theta).\log P(x_i|x_{i-1},\theta)$$
$$= -\frac{1}{\sum_{x\in X}|x|}\sum_{x\in X}\log P(x_i|x_{i-1},\theta)$$

For example, we are given the sentence "wipe off the __"
From our training data, assume we have:

the sweat = 1034
the dust = 800
etc.
----
the * = total count

- The value $\sum_{x\in X}|x|$ is the normalizer using the total count of "the *" that is used to compute per word probability.
- $P(x_i|x_{i-1},\theta)$ is the probability of the word (eg. sweat, dust etc) given the word "the".
- $\sum_{x\in X}\log P(x_i|x_{i-1},\theta)$ = sum of all log probabilities of the bigram "the __"

We use the above to compute the entropy.
Now, the units of entropy is bits and its not very useful. Instead, we use "**perplexity**".

$$\boldsymbol{perplexity\,(x,\theta) = 2^{H(x,\theta)}}$$

- A perplexity of k tells us that the model is as uncertain as if it had to choose from k elements with equal probability.
- Lower the perplexity the better the model.
- Best perplexity value is 1.
- It is a measure of how "confused" the model is.
- Examples for intuition:
  - If a task has to choose from 10 different digits (0,..10), perplexity = 10
  - If a task has to recognize 30,000 names, perplexity=30,000

Note:
- Easy to get bogus perplexities if probabilities are not normalized correctly.
- The average over actual words, not including START and STOP.

1.2.2.2 Word Error Rate (Extrinsic Evaluation)

$$WER = \frac{insertions + deletions + substitutions}{True\ sentence\ size}$$

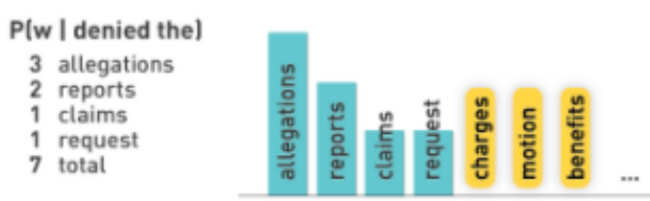
Example:
Correct Answer : Andy saw a part of the movie
Model output : And he saw apart of the movie

WER = (1 + 1 + 2) / 7 = 57%

1.2.3 N-Gram Smoothing

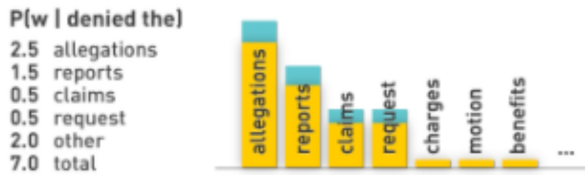
- With N-grams, as order increases, data gets more sparse.
- **Zipf law** describes relation between word types and tokens.
    - Word frequency is inversely proportional to word rank. (Where most common word has rank=1 and so on)
    - Power-law distributions like Zipf's law have a **long tail**, which means that a large fraction of the tokens (words on the page) belong to types (words in the dictionary) that appear quite rarely.
- Now, since our words follow the Zipf's law and our MLE is based on relative frequencies, we run into the problem of getting zero counts.
    - We need to avoid zero probabilities and have better estimates for rare words or n-grams.
    - Smoothing is a way to achieve this.

- **Smoothing**
    - With the regular MLE we use, below is how we would get a distribution for some sample data.

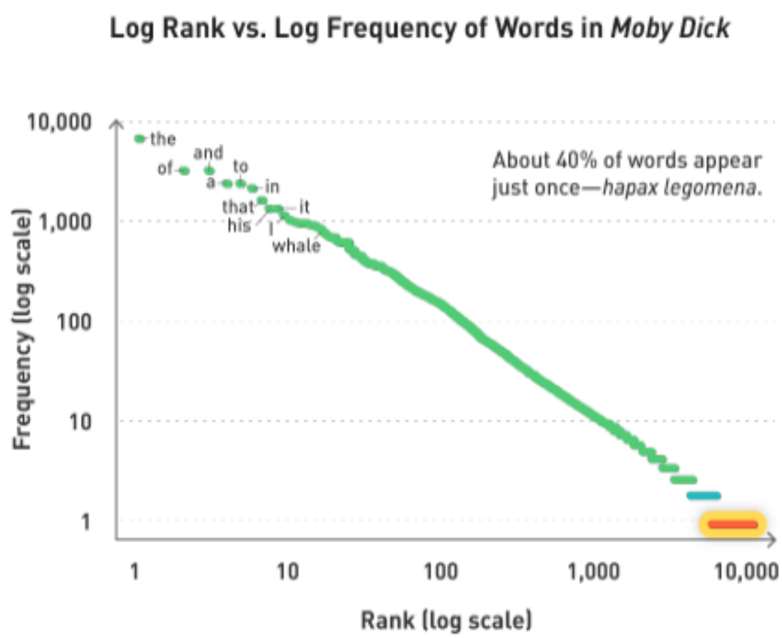


- While words in the training corpus are assigned some probability, unseen words are completely ignored.

- Smoothing tries to flatten the curve by distributing the weight such that unseen words are assigned some probability too.



- This problem of sparsity is very prevalent. Its so prevalent that words that appear just once in a corpus are called "Hapax Legomena"



1.2.3.1 Katz Smoothing

- Also called Backoff Smoothing
- In this approach, we use some predefined threshold, k.
  - If count(context) < k, reduce the size of context.
  - i.e. we look at trigrams. If trigrams are less than the threshold, we look at bigrams and then finally at unigrams.
    - Trigrams → bigrams → Unigrams

1.2.3.2 Interpolation

- This is an extension to Katz smoothing.
- Instead of using one of the n-grams at a time, this approach uses all.

$$\lambda\, P(w_t) + \lambda^{`}\, P(w_t|w_{t-1}) + \lambda^{``}\, P(w_t|w_{t-1}, w_{t-2}),$$
$$where\ \lambda + \lambda^{`} + \lambda^{``} = 1$$

- For example,

$$P(door|close, the) = \lambda\, P(door) + \lambda^{`}\, P(door|close) + \lambda^{``}\, P(door|close, the)$$

- There are reasonable ways to estimate λ, like using dev or held out data to compute hyperparameters.

1.2.3.3 Laplace Smoothing (Add δ smoothing)

- Add a small number to each n-gram count

$$p_{Laplace} = \frac{count(w)+\delta}{\sum_{w}[count(w^{`})+\delta]}$$

- For example, if vocab size = V and $\delta = 1$

$$p(sauce|the) = \frac{count(the,sauce)}{count(the,*)}$$

With Laplace smoothing,

$$p_{Laplace} = \frac{count(the,sauce)+1}{count(the,*)+V}$$

1.2.3.4 Good-Turing Smoothing (Held out reweighting)

This approach highlights the fact that

- Add 1 smoothing overestimates fraction of new n-grams
- Add very small δ underestimates fraction of new n-grams

This approach computes the fraction to use based on the count as shown below.

Count in 22M Words	Actual c^* (Next 22M)	Add-one's c^*	Add-0.0000027's c^*
1	0.448	2/7e-10	~1
2	1.25	3/7e-10	~2
3	2.24	4/7e-10	~3
4	3.23	5/7e-10	~4
5	4.21	6/7e-10	~5
Mass on New	9.2%	~100%	9.2%
Ratio of 2/1	2.8	1.5	~2

- Column 1 shows count of words appearing once, twice and so on in the training data.
- Column 2 shows the fraction of the count of the same words in a held-out data set.
- This is then used to compute the actual probability of the word or n-gram.
    - So instead of always using $\delta = 1$ or some fixed value like in Laplace, we end up with a value for δ that is between 0 and 1 depending on the word seen.
- Why is it that the fraction is always less in the next dataset?
    - The occurrence of a particular n-gram is rare. So, if it is seen in the training set, then it is less likely to be seen again in the held-out set.
    - So, observed n-grams appear more than they will later.

1.2.3.5 Handling Unknowns

What to do with *n*-grams that you never saw during training?

- Laplace: Assign a small probability to each one.
- Better: Lump them all into a single entry called "<UNK>"
- Best: Draw inference from the letters?

1.2.3.6 Absolute Discounting

- This is an extension to Good-Turing smoothing
- In the example from good-turing, shown below:

Count in 22M Words	Future c^* (Next 22M)
1	0.448
2	1.25
3	2.24
4	3.23

  - We see that there is almost the same difference of around 0.75 between count in training set and fraction in next set.
  - Using this fixed difference for smoothing is called absolute discounting.

- Absolute Discounting is defined as

$$p_{AD}(w|w`) = \frac{count(w`,w)-d}{\sum_w[count(w`)]} + \alpha(w`).\hat{P}(w)$$

Where,
  - d is the discount or re-distribution factor
  - $\alpha(w`)$ is the back-off factor.
  - $\hat{P}(w)$ is the back-off distribution.

1.2.3.7 Kneser-Ney Smoothing

When building an n-gram model, we're limited by the model order (e.g. trigram, 4-gram, or 5-gram) and how much data is available. Within that, we want to use as much information as possible. Within, say, a trigram context, we can compute a number of different statistics that might be helpful. Let's review a few goals:

1. If we don't have good n-gram estimates, we want to back off to (n-1) grams.
2. If we back off to (n-1) grams, we should do it "smoothly".
3. Our counts are probably *overestimating* for the n-grams we observe (see *held-out reweighting*).
4. Type fertilities tell us more about $P(w_{new}|context)$ than the unigram distribution does.

Kneser-Ney smoothing combines all four of these ideas.

1.2.3.7.1 Absolute discounting

- This gives us an easy way to back-off (1. and 2.) by distributing the subtracted probability mass among the back off distribution $\hat{P}(w)$.
- The amount to redistribute, δ, is a hyperparameter selected based on a cross-validation set in the usual way.
- For a trigram, we get:

$$P(c|b,a) = \frac{\max(0, count(a,b,c) - \delta)}{count(a,b)} + \alpha(a,b).\ \hat{P}(c|b)$$

- *Note:* we use max since we need the numerator above to positive

1.2.3.7.2 Type fertility

- The back-off model really doesn't give us good results.
- Instead, we need a word that that is allowed in a novel context.
- For example,
    - There was an unexpected ____. [delay or Francisco?]
        - Among unigrams, Francisco is more common than delay.
        - However, Francisco is almost always preceded by San.
        - So, it is less fertile than delay.
- So, for each word, count the number of bigram types it completes.
- We define the back-off distribution $\hat{P}(w)$ as proportional to the type fertility of w, i.e. the number of unique preceding words $w^{`}$

$$\hat{P}(w) \propto \left|w^{`} : count\left(w^{`}, w\right) > 0\right|,\ where, w^{`}$$ is the set of words preceding w

So, we define **type fertility(tf)** of w as:

$$\text{tf(w)} = \left|w^{`} : count\left(w^{`}, w\right) > 0\right|$$

- To make $\hat{P}(w)$ a valid probability distribution, we need to normalize it. This gives us:

$$\hat{P}(w) = \frac{tf(w)}{Z_{tf}}$$, where $Z_{tf} = \sum_{w^*} tf(w^*)$

1.2.3.7.3 KN Equations

Putting the above together, for a trigram we get:

$$
\begin{aligned}
P(c|b,a) &= \frac{\max(0, count(a,b,c) - \delta)}{count(a,b)} + \alpha(a,b).\ \hat{P}(c|b) \\
&= \frac{\max(0, count(a,b,c) - \delta)}{count(a,b)} + \alpha(a,b).\left[\frac{\max(0, count(b,c) - \delta)}{count(b)} + \alpha(b).\hat{P}(c)\right] \\
&= \frac{\max(0, count(a,b,c) - \delta)}{count(a,b)} + \alpha(a,b).\left[\frac{\max(0, count(b,c) - \delta)}{count(b)} + \alpha(b).\frac{tf(c)}{Z_{tf}}\right]
\end{aligned}
$$

Where,

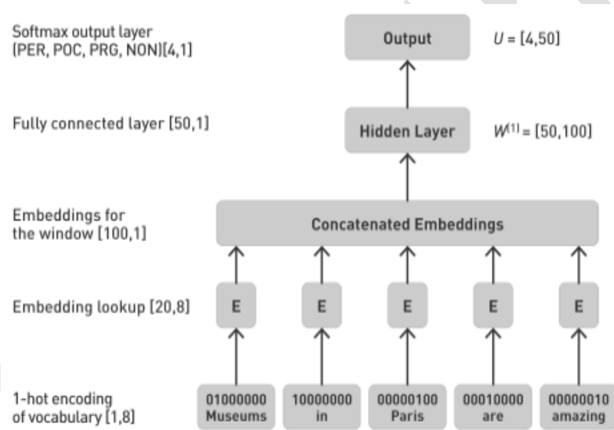
$$\alpha(b) = \frac{\delta}{count(b)} * count\left(c^{`} : count\left(b, c^{`}\right) - \delta > 0\right)$$

$$\alpha(a,b) = \frac{\delta}{count(a,b)} * count\left(c^{`} : count\left(a, b, c^{`}\right) - \delta > 0\right)$$

1.2.4 Neural Net Language Models

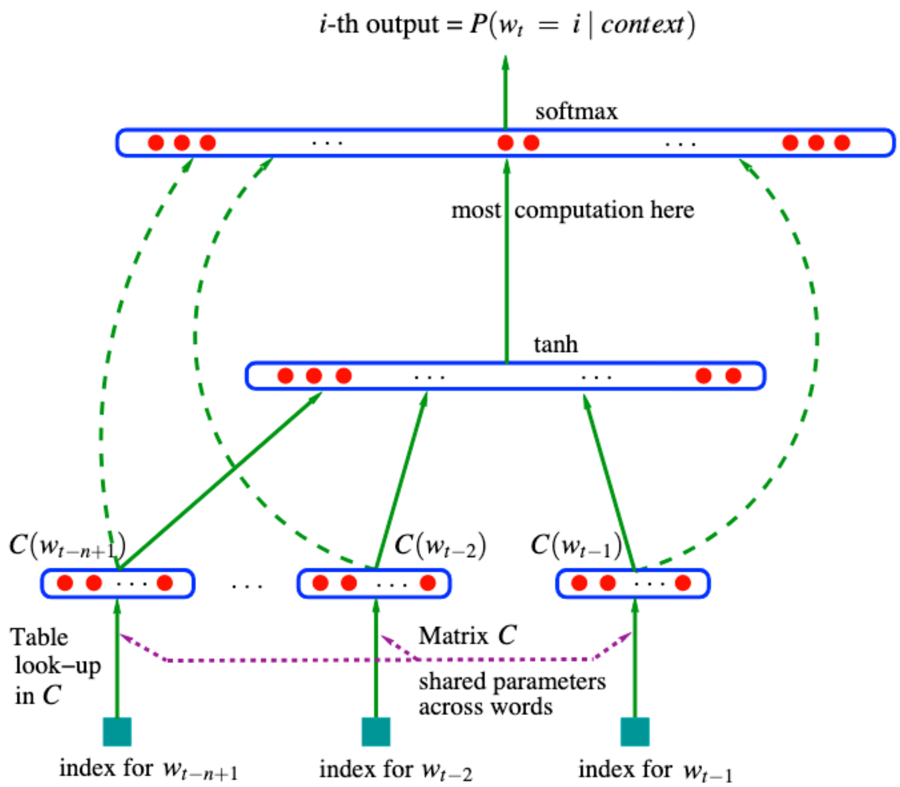
- In n-gram models
  - We use a lot of data and smoothing techniques
    - Higher order n-grams get better results
  - We have a sparsity problem.
  - Words are atomic
  - We use the Markov assumption which is not true.
  - We cannot model long range dependencies.
- Using neural nets,
  - We can predict the next word given some context. (our language model)
  - We use a vector representation of words.
  - We use hidden layers to reduce dimensionality of the context
  - We can make the model efficient to train with lots of data.

Below is a sample network:



- Input is an array of words.
- Each word has a vector representation in the word embeddings.
  - In the above example, its 1 hot encoded.
- We look up the word representation from the embeddings and concatenate the resulting outputs.
- This concatenated vector is fed as input to the hidden layers.
- The hidden layers go through their activation functions.
- The final output is then fed to a softmax layer to obtain the required number of predicted classes.

1.2.4.1 NPLM: [A Neural Probabilistic Language Model, Bengio et al]



In an n-gram model of order k,

$$P(w_t|w_{t-1}, \dots, w_0) \approx P(w_t \mid w_{t-1}, \dots, w_{t-k})$$

Where, the estimated probabilities were smoothed maximum likelihood estimates.

For the NPLM, we'll replace that estimate with a neural network predictor that directly learns a mapping from contexts $(w_{t-1}, \dots, w_{t-k})$ to a distribution over words w_t.

$$P(w_t \mid w_{t-1}, \dots, w_{t-k}) = f(w_t, (w_{t-1}, \dots, w_{t-k}))$$

Broadly, there are three parts:

1. **Embedding layer**: map words into vector space
2. **Hidden layer**: compress and apply nonlinearity
3. **Output layer**: predict next word using softmax

- The model also has *skip connections* between the embedding layer and the output layer.
    - This just means that the output layer takes as input the concatenated embeddings in addition to the hidden layer output.
    - This was considered an unusual pattern, but has recently become popular again in the form of Residual Networks and Highway Networks.

- **Notation**
    - **Hyperparameters**
        - V: Vocab size
        - M: Embedding size
        - N: Context Window size (i.e. number of words in input)
        - H: Number of hidden units or neurons

    - **Inputs**
        - Ids_ : (batch_size, N), integer indices for context words
        - y_ : (batch_size,), integer indices for target word

    - **Parameters**
        - Embeddings matrix, C_ : (V,M)
        - Hidden Layer, W1_ : (NxM,H)
        - Bias, b1_ : (H,)
        - Output or softmax Layer, W2_ : (H,V)
        - Matrix for skip layer conns, W3_ : (NxM,V)
        - Bias, b3_ : (V,)

    - **Intermediate states**
        - x_ : (batch_size, NxM), concatenated embeddings
        - h_ : (batch_size, H), hidden state = $\tanh(x\_.W_1 + b_1)$
        - logits_ : (batch_size, V), = $h\_.W_2 + x\_.W_3 + b_3$

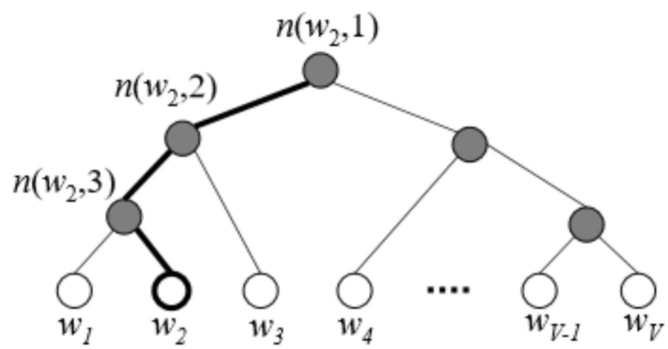
1.2.4.1.1 Hierarchical Softmax

In the NPLM, most of the computation is at the softmax layer, because output is normalized over the entire vocabulary.

- We compute the probability of every word in the vocab
- This is then used to normalize the output scores and obtain the argmax.
- Computational complexity is O(V).

In hierarchical softmax,

- We use a tree structure to compute the normalized probability and this reduces the complexity to $O(\log_2(V))$.



- In this structure, every word in the vocab is a leaf node.
- Now, in the NPLM model, every word has an input and output representation.
    - As an example, let's say we use 1-hot encoding to represent words.
        - If our vocab is V, each word is represented by a vector of size Vx1
        - Now assume we have N hidden layers.
        - Our output from the hidden layers would be a matrix of size VxN given by:

$$h = W^T . C_i := v_{w_i}^T$$
$$where$$

W is the weight matrix and

C_i is the vector representation of word, w_i

        - "h" above is the **input vector representation** of the word,
            - Formally, row i of the weight matrix is considered the input vector representation of a word, w_i.
        - Let the **output vector representation** be represented as $\grave{v}_{w_i}$.

- Now, each of the words can be reached by a path from the root through the inner nodes, which represent probability mass along that way.
- The probability mass of a path is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Where,

x is the dot product of the input and output representations of the word.

$$x = \left(\grave{v}_{n(w,j)}\right)^T . h_{w_i}$$

Where,

- w is the target word
- j is the j-th node on the path from root to w
- w_i is the input word or context
- h_{w_i} is the input vector representation of word w_i
- $\grave{v}_{n(w,j)}$ is the output vector representation
- $n(w,j)$ is the j-th node from the root on the path from root to word w

- We can now compute the probability at node n, branching left or right given state **h** as:

$$p(n, left) = \sigma\left(\left(\grave{v}_n\right)^T . h\right)$$
$$p(n, right) = 1 - p(n, left) = \sigma\left(-\left(\grave{v}_n\right)^T . h\right)$$

- Now we use the above to compute the word probabilities

$$p(w|w_i) = \prod_{j=1}^{L-1} \sigma(\langle n(w, j+1) == child(n(w,j)) \rangle \left(\grave{v}_n\right)^T . h)$$

Where,

- L is the height of the tree
- Angle brackets represent boolean checking

- In our example tree, for word w_2 we get:

$$p(w_2|w) = p(n(w_2, 1), left) . p(n(w_2, 2), left) . p(n(w_2, 3), right)$$

$$= \sigma\left(\left(\grave{v}_{n(w_2,1)}\right)^T . h\right) . \sigma\left(\left(\grave{v}_{n(w_2,2)}\right)^T . h\right) . \sigma\left(-\left(\grave{v}_{n(w_2,3)}\right)^T . h\right)$$

- **Constructing the tree**
    - Morin and Bengio used WordNet to build the tree:
        - Used WordNet synsets and hypernym (is-a) relationships
        - Converted the WordNet synset hierarchy into a binary tree
    - Subsequent work experimented with other methods:
        - Randomly constructed trees
        - Brown clustering
        - Huffman binary encoding (based on word frequency)

1.2.4.2 Recurrent Neural Nets (LSTM)

- No Markov assumption: Since long range dependencies can be handled.
- In 2016, LSTMs beat n-gram based LMs.
- Few adjustments were made:
    - Approximated output softmax with a random sample of words in the vocab.
        - This called **importance sampling**
    - Added a linear projection layer before the softmax to reduce number of parameters
    - Stacked two LSTMs: output from first LSTM is input to the second.
    - Used Adagrad instead of Stochastic Gradient Descent
    - Initialized forget gate biases to 1: That is, keep everything.
    - Parameters were limited to what could fit in a single GPU memory.
    - Used distributed computing with 32 GPU workers.
        - Used asynchronous gradient updates.
    - Got perplexity of score of around 30.6 compared to a score of 51 for n-grams.

End of Document