Automatic Speech Recognition

Mostafa ElFaggal s2653879
Rana ElGahawy s2650721

Abstract—This paper aims to build a speech recognition system. It is based on HMM-DNN modeled by WFSTs. The model is designed only to operate on a limited lexicon constructed from the famous tongue twister. In the first section, we build a baseline for the model. All further sections aim to improve the model, either on a run-time basis or an accuracy basis. There are 329 recordings on which the model is trained and tested. The final results are that the model runs with an accuracy of **36.8%** and a run-time of 5 minutes and 50 seconds when run sequentially and parallelly, respectively.

1 Introduction

This paper aims to build a speech recognition system. It is based on HMM-DNN modeled by WFSTs. The model is designed only to operate on a limited lexicon constructed from the famous tongue twister. The tongue twister goes as follows: P*eter Piper picked a peck of Pickled Peppers. Where's the peck of pickled peppers Peter Piper picked.* The words used may be of any combination and in any random ordering (grammatical rules are ignored). In the first section, we build a baseline for the model. Next, we focus on tuning the model, such as adjusting the weights and introducing a Unigram language model. We also include a silence model to better fit the neural network's design. Other time-based improvements were also performed, such as Pruning. Then, there are some advanced improvements, including a tree-structured lexicon, look-ahead pruning, and using a Bigram language model. Beyond these improvements, there were extra tweaks, such as caching the Observation probabilities and parallelizing the running tests. All further sections aim to improve the model, either on a run-time basis or an accuracy basis.

2 Initial System

For the first task, our main goal was to build a Viterbi decoder that would link the produced probabilities from the Neural Network to an HMM. As such, we would be able to decode and produce some predictions. The HMM, developed using the Open-WFST library in Python, was designed to have the phone states as inputs and entire words as output. Fig [1] 1 represents the FST used. There is a starting node that expands with equal probabilities to all other starting nodes of each word (these transitions don't consume any symbols). From there, a linear FST consuming phones, where each phone is split into 3 states, producing the word only at the last step. Epsilons are produced along the way until the word is produced. After the word is produced, the last state of the word is considered a final state, as the prediction has to end with a word. Next, the states return to the starting state to repeat the process for multi-word prediction. All phone state nodes have self-loops to account for more than 3 frames per phone. Probabilities are set to 10% for self loops and 90% $(100\% - 10\%)$ for traversing forward. This means that the model favors producing more words rather than fewer.

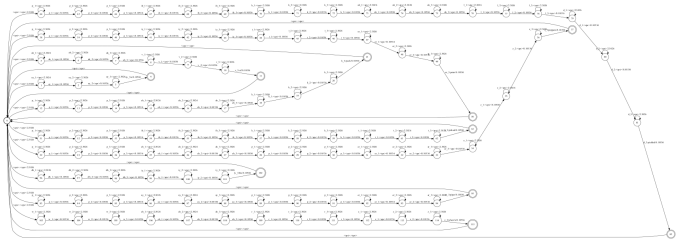

Fig. 1. Full WFST created for the initial baseline that support predicting words from phones in a cycle

After building our baseline, we started testing its accuracy and speed. The model ran in **28 mins** within a word error rate (WER) of **144.88%**. The **WER had 605, 19, and 2885 for Substitutions, Deletions, and Insertions (S, D, I), respectively.** As noticed from the high I and low D, the model was overly predicting, tending to make longer predictions. Consequently, one of the improvements for this model could be increasing the self-loop probabilities, hence nudging the model to spend more frames processing each word, thus improving I.

From this point forward, we moved from using our local machines for the development

1. All the figures are available in this Google Drive

to the DICE machines in the labs which significantly affected the runtime from 28 mins to 8 mins.

Currently, the baseline model is working functionally, the next sections aim to improve its run time and accuracy in terms of WER.

3 System Tuning

One crucial step after building any model is to tune its hyperparameters to improve the system's functionality and accuracy. in the context of our Speech Recognition system, we considered three different methods to achieve better results while tuning the system which are:

- Silence states
- Unigram Language Model
- Weight adjustment

3.1 Silence States

The first improvement to the model is accommodating the possibility of having silence states between the words by adding a silence model to the baseline WFST.

We used the same silence topology (Fig 2) on which the neural network was trained, which includes 5 states in which state 1 and state 5 only have a self-loop and one forward traversal arc to the following state. Meanwhile, the in-between states have an Ergodic structure.

The model was designed such that the last state of each word would have an arc to the initial state of the silence model. An unweighted arc was also added from the last state in the silence model to the starting state of the entire WFST to keep the cycle of prediction going. This model was initially implemented to account for the silences mid-sentence.

The probabilities on the silence model's states 1 and 5 follow the same pattern used for the words with **10%** probability for the self-loop and **90%** probability for traversal arcs. However, the rest of the states (2,3,4) follow an Ergodic model. Their probabilities are equally distributed between all the arcs emerging from each state.

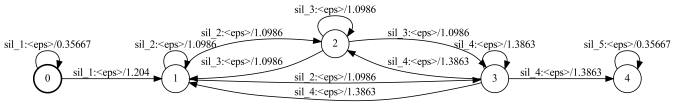

Fig. 2. Silence states added to the WFST to match the neural network

The model was significantly improved after adding the silence model as the **WER** became **81%**, which is **1.8 times better than our initial baseline**. At the same time, there was no significant change in the model's runtime. At later stages in the improvement of the model, the last state of the silence model was made a final state to consider the silences at the end of the recordings, which further improved the **WER**. See section 3.4

Silences at the start of the recordings were also later considered by adding an arc from the initial state of the model to the first state in the silence model. Consequently, it improved the **WER** further. See section 6.4

Results and Discussion:

The neural network used in the model was trained to take into account the possibility of having possible silent states between words. Before adding the silence model, the neural network was always expecting the presence of some silence states to exist within the frames of the recordings; however, the WFST didn't accommodate such behavior, forcing the model to add extra words instead, resulting in the over-prediction produced by the baseline and high Insertions. The addition of the silence model made a huge impact on that. Currently, the neural network and the WFST are compatible with each other, and both accommodate for the presence of the silence states, making a significant change in the WER calculated.

Adding the silence model made the **WER** reach **81%** after it was **144.88%** in the baseline. The **WER's SDI** dropped down to **622, 40, and 1302** respectively.

3.2 Unigram LM

At this stage, a Unigram language model was introduced in addition to the WFST. The lan-

Word	Start Freq	Freq	End Freq
a	25	138	0
of	4	246	5
peck	20	264	21
peppers	12	321	119
peter	121	338	28
picked	10	286	98
pickled	28	285	19
piper	34	289	37
the	17	137	0
where's	58	118	2

TABLE 1: Frequency of words at start, mid, and end of utterances based on the true subscriptions

guage model was designed to include the frequencies of the existence of all the words at the start, anywhere, or end of the utterances, as shown in Table 1. The frequencies were extracted from the true transcriptions of our training dataset. Although this produces potential issues of over-fitting, the aim was to adjust to the somewhat random nature of the recordings since not all are proper English sentences. Further, normal language models that take into account the entire English language wouldn't be a good fit for the given case of only the tongue twister's lexicon.

These frequencies were used to adjust the weights in the WFST for each word being at the start, end, or anywhere in the predicted transcription.

We tried different combinations of these frequencies:

- start, mid, end
- start, end
- mid, end

The end frequencies were used in two main methods. One is to account for the final state weight, simply the frequency of the predicted word being the last word in the recording. Another use was the arcs traversing back from the last state of every word to the start state (so as to continue predicting). For that case, we use 1-prob final. That is to say that the probability of returning to the start to predict further is complementary to being the last word in the utterance.

As for the transition from the start state to the beginning state of every word, there were different possible probabilities to use. One example we tried was to set them to start frequencies. However, we also considered using the mid frequencies as any further predicting beyond the first word would make use of said frequencies, and the mid probabilities are more reasonable. A final combination was to split the starting state into two states: the init state and the start state. The init state is the actual starting state of the WFST, and the arcs heading from it to the words are weighted by the start frequencies. The word endings would then traverse back to the start state, which would traverse to the word beginnings using the mid frequencies.

Results and Discussion:

In terms of results, the data was mostly consistent. The run-time was unchanged. However, the accuracy shifted slightly around. Using only the start frequencies, the accuracy dropped to **78%**. Using the mid frequencies only resulted in **88%**. Making use of both frequencies also yielded **88%**. We believe that the 78% is more of a fluke than anything. As for the mix frequencies, their output is minimally variant due to the overshadowing effect, which will be discussed in section **3.3**. Finally, the reason for using a mix of frequencies and using only the mid frequencies turned exactly the same accuracy and WER SDI is that the mid frequencies made the starting frequencies insignificant. The starting frequencies are only consumed once at the beginning of the prediction path. Meanwhile, the mid frequencies are constantly used to predict every other word. As such, whether or not we have this one extra frame at the beginning, using start vs. mid frequencies, would be irrelevant to the bigger picture, and as such, the accuracies were identical. We decided to continue using the mix frequencies model, which made the most sense. The hope is that

further improvements will work well with this more reasonable model.

3.3 Weight Adjustments

Regarding changing the weights, there were two main components: self-loops and the language models overshadowing effect. The idea for the self-loop probabilities is that lower self-loop probabilities encourage the model to forward traverse more and produce more words in the transcription. Conversely, higher self-loop probabilities nudge the model towards staying within a word for more frames and, thus, overall, produce fewer words. The baseline model used self-loop probabilities of **0.1**, which is a low amount and is consistent with the theory when noticing the high amount of Insertions in the WER.

The other major factor to tweak was related to the overshadowing effect. The Unigram language model probabilities produced from the previous section were overshadowed by the probabilities of the self-loops and the forward traversals within every word. We name the weights referring to the language model's external probabilities weights. On the other hand, we label the self-loops and forward traversals within a word's internal weights. Consider the case where one path has a summed-up weight (NLL domain) of 100, and another has a weight of 200. Irrespective of the frequencies of the Language Model (LM), the best path had already been chosen based on the internal weights. The reason the internals are more prominent than the externals is that there are more internal arcs than there are externals. In the probabilities domain, where they are being multiplied, there are more chances for the overall value to drop to very small probabilities very quickly. Identically, in the NLL domain, there are more arcs for internals and, hence, more chances to quickly add up to large numbers, which overshadow the effects of the LM.

The LM isn't designed to be the main focus of predictions; rather, its main job is to avoid the more unlikely words while giving space for the neural network to do its magic and predict the words but in a more confined and likely word space. The solution to the overshadowing issue is either to deafen the internal weights or louden the external weights. Reducing the internal probabilities leads their NLL weights to grow, and thus, the model would avoid picking them and instead predict based on the LM. The opposite, however, is that where we loud-en the external weights, the bad word choices are even more noticeable not to pick. As such, we decided to enlarge the external NLL weights. To enlarge the weights, there were multiple options: adding a constant bias, multiplying by some constant, apply some constant exponent. The method that would make the bad words stand out the most would be exponentiation, as the gap between the external weights would be widened the most.

To decide on the best self-loop probability and the best constant for exponentiation, we ran tests with various values shown in Fig 3.

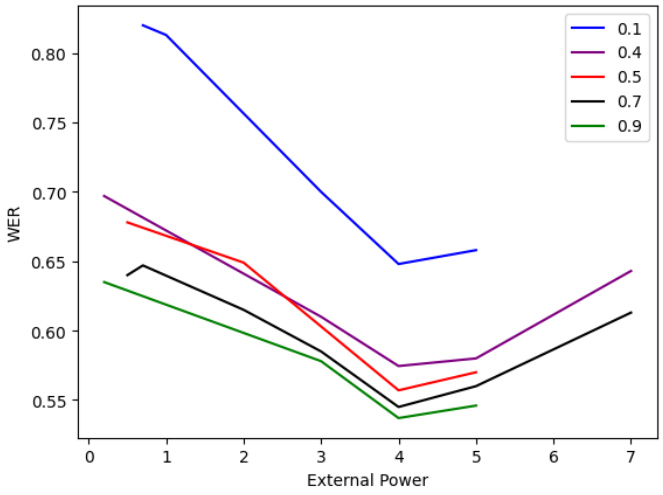

Fig. 3. The effect of having different self loop probabilities on External Weights versus WER

Results and Discussion:

A clear example of diminishing returns can be seen in the various external weight exponents. The internals cloud low exponents and the external weights, while too-high of weights and the LM take over the prediction process. A similar case could be expected from the self-loop probabilities. As shown in Fig 3, the lowest **WER** of **53.7%** was achieved with a self-loop with probability **0.9** and a **4** External power.

However, since it is at the extreme of the self-loop, it might be over-fitting to this specific data set and probably just a fluke. The choice was made to have the self-loops with probability **0.7** and External Power of **4** with a **WER** of **54.5%**. It wasn't a significant change from the lowest WER achieved, but at the same time, it is more generic and on the safe side of the diminishing returns case. This significant decrease in the **WER** is mainly because of the weight introduced by the Unigram LM introduced in the previous section; previously, they were overshadowed by the internal probabilities of each word, but as now they have high powers, their presence is more seen when calculating the overall probability this word, consequently improving the predictions and the WER.

3.4 Silence Ending

One simple idea we hadn't considered in our first iteration of the silence model is to set the last state of the Silence model to be a final state. This allows for silence at the end of a recording.

Results and Discussion:

This made a drop in the **WER** as it improved to **53%**. This improvement is expected as in some recordings, the person pauses before stopping the recording.

4 Pruning

Pruning is a time-based improvement. In a nutshell, any path whose probability drops beyond a certain threshold of the current best candidate path can be ignored so that overall speed gain can be attained. This approach is heuristic and thus may lead to a drop in accuracy. We decided to test different threshold limits. Extremely harsh pruning may lead the algorithm not to find a valid path, as all possible valid paths have been pruned early on. Conversely, a very high threshold loses sight of the goal of pruning. Just like the weights choosing the pruning threshold for our model, we started testing different thresholds (Table 2) until a proper value was achieved that didn't affect the **WER** significantly but introduced a noticeable decline in the runtime.

Threshold	WER	Run Time
5	No Valid Path	
7	No Valid Path	
10	60 %	42 s
20	56 %	47 s
30	54 %	50 s
40	53.6 %	53.5 s
50	53.3 %	57.4 s
No Pruning	53%	1.22 mins

TABLE 2: The effect of different Pruning Thresholds on the WER and the Run Time

Results and Discussion:

As expected from how pruning works, as we increase the threshold, the WER and the runtime decreases. While it is better to have a smaller runtime, we still need to account for the **WER** as it would be rather better to have a lower WER than a lower time. Further, too harsh of pruning can lead to Invalid paths, as seen with thresholds 5 and 7. Being too close to those harsh thresholds is equally dangerous, as although it works for the current dataset, it may not work for other data points, given how harsh the cutoff is. Consequently, we chose 40 as our threshold as the time declined to **50 seconds** (dropping by around 30 seconds) while the WER increased, and we believe that 40 is a safe enough margin from the harsh cutoffs. It is worth noting that while recording these data, the model was running in parallel on multiprocessing, which is explained in detail in section 6.2. However, after fixing the threshold, the model was tested while disabling the parallelism, and it was found to have improved from **8 mins** to **5 mins**, which is a significant drop and shows that our selected threshold has a notable effect on the runtime.

5 Advanced Topics

Various potential improvements were considered. Some were time-based, such as Look Ahead and tree-structured lexicon. While others are accuracy-based, such as Bigram LM.

5.1 Tree-Structure

The idea of the tree-structured lexicon is to coalesce similar phones to minimize the WFST. We noticed that the Viterbi Algorithm is roughly running in O(T * S * S * Ob), where T is the number of frames, S is the number of States, and Ob is the complexity of the observation model computing its probabilities. The algorithm computes for every time frame, every arc (upwards of S*S), an observation probability. We improved on the Ob portion in section 6.1 Observation Caching. As for T, there isn't much that can be done on the number of frames in a recording. A tree-structured lexicon focuses on minimizing the S. This approach is similar to Trie trees, except we use Phones instead of letters. Further, coalescing can be done on both the word beginnings and the word endings.

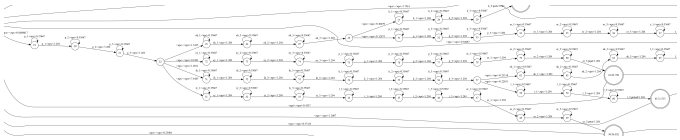

Fig. 4. Part of the tree structure for the words starting with phone P

Results and Discussion:

However, after doing it by hand, we noticed that only front coalescing is fruitful, given our limited vocabulary. Fig 4 is the model after applying front coalescing. To redistribute the LM's external weights, we first decided to place collective frequencies at the head of every tree branch, while conditional frequencies were placed at the branching points of the tree.

For the simplicity of the symbols, we'll show the equations we used in terms of Peter and its first phone P to calculate the probability at Peter:

$$\mathcal{P}(Peter|P) = \frac{\mathcal{F}(Peter)}{\Sigma\mathcal{F}(P)} = \frac{\mathcal{P}(Peter)}{\Sigma\mathcal{P}(P)}$$

The idea is that after traversing the entire word's branch, the same results would be attained as those of the non-tree structure. The accuracy remains unchanged while the run time is improved. On the other hand, this readjustment to the external weight caused us to drop the usage of the two separate init and start states and the start/mid/end combo frequencies of the LM (see section 3.2). This is because weight balancing wasn't possible in a tree structure unless we duplicated the tree so that each tree could use its respective weights; only this would cause more states to miss the point of a tree-structured lexicon. However, since the start frequencies didn't contribute to the mix frequencies model as described in section 3.2, the changes that the tree structure imposed weren't serious. This adjustment made the model use from the language model the frequencies of each word being at the end of the sentence or merely the existence of the word in the sentence. Overall, the number of states was reduced from 132 to 106, which isn't a significant enough improvement to be noticed. In fact, run time-wise, no effects could be noticed.

5.2 Bigram

We improved the LM, by adding frequency computations that take into consideration the previous word. These new frequencies can be used by adding an arc from the end of every word to the starting state of every word(s), weighted by those Bigram frequencies.

Results and Discussion:

After running the model, no improvements could be noticed. Neither accuracy nor runtime changed. We believe this is due to the tree structure that we were using. To accommodate the tree structure, the Bigram probabilities were combined. So, for example, the arc from the last state of "of" to the start state of the branch of "a" and "of" has a combined probability of $P(a|of) + P(of|of)$. Although $P(of|of) = 0$, $P(a|of)$ is high, and so $P(a, of|of)$ is also high. In a non-tree structure, the transcription "of of" would be impossible. However, due to the tree structure enforcing the collective Bigram probabilities, the true power of Bigrams is lost, and cases such as "of of" become more common rather than being impossible.

5.3 Look ahead

Another time-based improvement is Look-ahead, where we can prune earlier. The idea is, while ensuring all weights are positive, push them as early in the tree as possible. As such, pruning would be able to make a more educated guess earlier on and hence shave off more run-time.

Results and Discussion:

After applying the look-ahead approach on the WFST's tree, the **Runtime** decreased to **50s**, and although there shouldn't be any accuracy improvements, the **WER** also decreased to **44.3 %**. This could be explained by the technique used by this model to readjust the weight and increase the effect of the frequencies extracted from the language model, as now the external probability at each node is larger than what it used to be, and since it is raised to an exponential, any minor change in it can have a significant effect on the resulted **WER**.

6 Extra Improvements

To further improve the run-time and the accuracy of the model, we considered different approaches. A key factor for time-based improvements is parallelism so as to leverage the hardware fully.

6.1 Observation Caching

As mentioned in section 5.1, the main bottlenecks of the Viterbi algorithm lie in computing the observation probabilities (Ob) and the number of arcs. To improve on Ob, one idea is to cache the computed probabilities so as not to recompute them. However, ultimately, minimum to no effects could be noticed from this step.

6.2 Parallelism

To improve the run-time of the model to its fullest. We decided to try splitting the work across the hardware. At first, we tried multi-threading. However, since the work isn't IO or blocking, multi-threading failed to be of value. Instead, we used multi-processing, which leverages the CPU cores and splits the work across them. To make it work, we had to reinitialize a neural network for every process (called nnet in observation_model.py). This may be considered a liability as the initialization is the second biggest bottleneck after decoding. Although there may be ways to improve upon them further, they were too much of an overhead for very little returns. As it is multi-processing, it reduced the run-time from 8 minutes to 1 minute 20 seconds. This later, when paired with pruning and the look-ahead, was further improved to its final run-time of 50 seconds. When it came to multi-processing, there was a parameter of how many tests should be run per process. On the one hand, the fewer the tests per process, the more parallelism there is, which is better. On the other hand, too few tasks per process and the backlash from the nnet initialization would be very noticeable. We tested with various process sizes (number of tests per process) and got the following results (Fig 5).

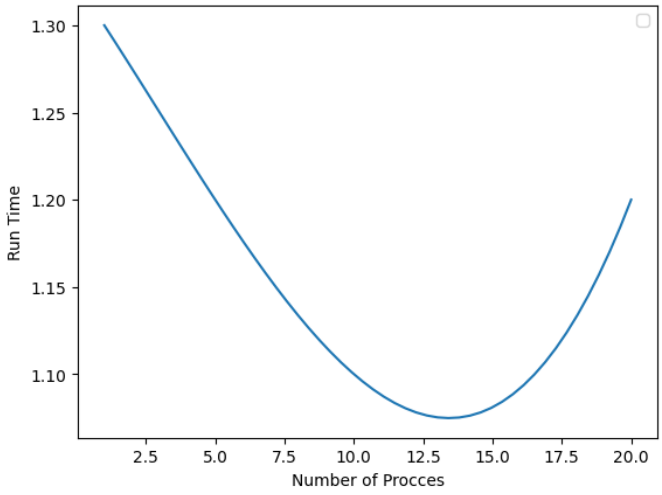

Fig. 5. The effect of increasing the number of processes on the runtime

6.3 No Word Silence

Another simple idea concerning the Silence model is to have an arc directly from the start state to the silence model's start state. This allows for no-word silences, where no word is needed for silence.

Results and Discussion:

The last addition to the model also created a significant improvement to the **WER** as it significantly dropped from **44.3%** to **36.8%**, which is the best **WER** achieved by the model. There was no change at all in the runtime as it remains **50s / 5 mins** using parallelism and single process, respectively.

7 Conclusion

Overall, many improvements were applied to the speech recognition model. The model is first created by a WFST to model a baseline. It had an accuracy of **144%** and a run-time of 30 minutes on a local machine. Then, we added the silence model to accommodate them during mid-transcription, which drops the accuracy to **81%**. From this point forward, we opted to use Dice Lab machines, which improved the run-time to 8 minutes. The Unigram LM came next to consider what words are probable and what aren't. This, however, due to overshadowing, didn't contribute positively; rather, the accuracy rose to **88%**. Afterward, there was observation model probability caching to speed things up. However, it didn't help improve it. At this point, we chose to implement the tree-structured lexicon, which again didn't affect the model positively or negatively. To truly speed up the model, we applied multi-processing, which dramatically bumped up the model's run-time to **1 minute 20 seconds**. Next, we did weight adjustments, where we tested a variety of different self-loop weights and external exponents. This adjustment dramatically improved the accuracy to **54.5%**. End silences came next again, pushing the accuracy to **53%**. The other time-based improvement, pruning, helped push the run-time down to **50 seconds** but increased the accuracy to **53.6%**. We then decided to improve the LM to be a Bigram one, which didn't have any effects on the model. We then applied look-ahead, which, in conjunction with the external exponentiation, helped improve the accuracy to **44.3%**. Finally, the no-word silence was added, achieving the final results. The final model has an accuracy of **36.8%**, a run-time with parallelism of **50 seconds**, and a run-time without parallelism of **5 minutes**. Further, improvements may be possible, such as dynamic pruning and dynamic weighting based on recording length or variability. Removing the tree structure, coupled with the Bigram LM, might also yield positive results. Below is the model's final figure.

Fig. 6. The model's final structure