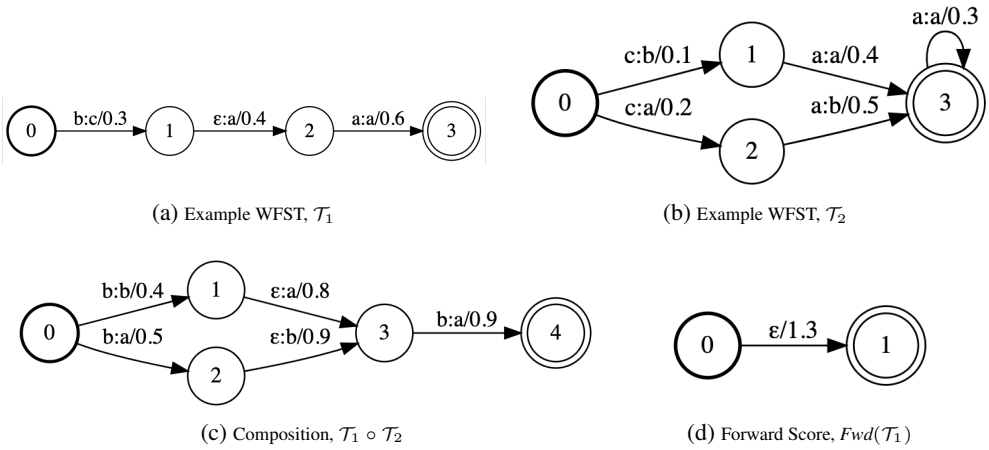
# A Appendix: Weighted Finite-State Transducers

A weighted finite-state transducer (WFST) is a generalization of a finite-state automaton (FSA) [29, 30, 39] where each transition has an input label from an alphabet $\Sigma$ an output label from an alphabet $\Delta$ and scalar weight $w$. Figure 10a shows an example WFST with nodes representing states and arcs representing transitions. A path from an initial to a final state encodes a mapping from an input sequence $\mathbf{i} \in \Sigma^*$ to an output sequence $\mathbf{o} \in \Delta^*$ and a corresponding score. While WFST operations can be performed in any semiring, in this work we only use the log semiring.




(a) Example WFST, $\mathcal{T}_1$ (b) Example WFST, $\mathcal{T}_2$

(c) Composition, $\mathcal{T}_1 \circ \mathcal{T}_2$ (d) Forward Score, $Fwd(\mathcal{T}_1)$

Figure 10: Examples of WFSTs and their operations (in log semiring). The arc label "$p$:$r$/$w$" denotes an input label $p$, an output label $r$ and weight $w$. Special symbol, $\epsilon$ allows to make a transition without consuming an input label or without producing an output label. Start states are represented by bold circles and final states by concentric circles.

In this work we use composition and shortest distance operations on WFSTs. The composition operation combines WFSTs from different modalities. Given two WFSTs $\mathcal{T}_1$ and $\mathcal{T}_2$, if $\mathcal{T}_1$ transduces $\mathbf{a}$ to $\mathbf{b}$ with weight $w_1$ and $\mathcal{T}_2$ transduces $\mathbf{b}$ to $\mathbf{c}$ with weight $w_2$, then their composition $\mathcal{T}_1 \circ \mathcal{T}_2$ transduces $\mathbf{a}$ to $\mathbf{c}$ with weight $w_1 + w_2$. The forward score operation is the shortest distance from a start state to a final state in the log semiring. Given a transducer $\mathcal{T}_1$, the forward score is the log-sum-exp of the scores of all paths from any start state to any final state. The example output graphs from composition and forward score operations are shown in Figure 10c and Figure 10d respectively.

## A.1 Autograd with WFSTs

Most operations on WFST are differentiable with respect to the arc weights of the input graphs. This allows WFSTs to be used dynamically to train neural networks. Frameworks like GTN [15] and k2 [19] implement automatic differentiation with WFSTs. For the purpose of this work, we use the GTN framework.

# B Appendix: Additional Implementation Details

## B.1 Automatic Speech Recognition

We keep the original 16kHz sampling rate and compute log-mel filterbanks with 80 coefficients for a 25ms sliding window, strided by 10ms. All features are normalized to have zero mean and unit variance per input sequence before feeding them into the acoustic model. We use SpecAugment [34] as the data augmentation method for all ASR experiments.

### B.1.1 Model architecture

The acoustic model (AM) architecture is composed of a convolutional frontend (1-D convolution with kernel-width 15 and stride 8 followed by GLU activation) followed by 36 $\times$ 4- heads Transformer blocks [38] with relative positional embedding. The self-attention dimension is 384 and the feed-forward network (FFN) dimension is 3072 in each Transformer block. The output of the final Transformer block if followed by a Linear layer with output dimension of 580 and a letter-to-word encoder (see Appendix C) to the output classes (word vocabulary + *blank*). For all Transformer layers, we use dropout on the self-attention and on the FFN, and layer

drop [12], dropping entire layers at the FFN level. We apply LogSoftmax operation on each output frame to produce a probability distribution (in log-space) over output classes. The model consists of 70 million parameters and we use 32 $\times$ Nvidia 32GB V100 GPUs for training.

### B.1.2 Decoding

**Beam-Search decoder**: In our experiments, we use a beam-search decoder following [8] which leverages a $n$-gram language model to decrease the word error rate. The beam-search decoder outputs a transcription $\hat{\mathbf{y}}$ that maximizes the following objective

$$\log P(\hat{\mathbf{y}}|\mathbf{x}) + \alpha \log P_{LM}(\hat{\mathbf{y}}) + \beta|\hat{\mathbf{y}}| \tag{5}$$

where $\log P_{LM}(.)$ is the log-likelihood of the language model, $\alpha$ is the weight of the language model, $\beta$ is a word insertion weight and $|\hat{\mathbf{y}}|$ is the transcription length in words. The hyperparameters $\alpha$ and $\beta$ are optimized on the validation set.

We use 5-gram LM trained on the official LM training data provided with LibriSpeech for beam-search decoding.

**Rescoring**: To further decrease the word error rate, we use the top-$K$ hypothesis from beam search decoding and perform rescoring [37] with a Transformer LM to reorder the hypotheses according to the following score:

$$\log P(\hat{\mathbf{y}}|\mathbf{x}) + \alpha \log P_{LM'}(\hat{\mathbf{y}}) + \beta|\hat{\mathbf{y}}| \tag{6}$$

where $\log P_{LM'}(.)$ is the log-likelihood of the Transformer LM, $\alpha$, $\beta$ are the hyperparameters as described above and $|\hat{\mathbf{y}}|$ is the transcription length in characters. We use the pre-trained Transformer LM on LibriSpeech from [24] which has a perplexity of 50 on dev-other transcripts. In this work, top 512 hypothesis from beam search decoding are used as candidates for rescoring.

### B.1.3 Optimizer

For training the word-based STC models, we use the Adagrad optimizer [10] with a learning rate warmup scheme that increases linearly from 0 to 0.02 in 16000 training steps for all the experiments. We halve the learning rate initially after 400 epochs and every 200 epochs after that. All models are trained with dynamic batching with a batch size of 240 audio sec per GPU. We use SpecAugment [34] with two frequency masks, and ten time masks with maximum time mask ratio of $p = 0.1$, the maximum frequency bands masked by one frequency mask is 30, and the maximum frames masked by the time mask is 50; time warping is not used. SpecAugment is turned on only after 32000 training steps are finished. Dropout and layer dropout values of 0.05 is used in the AM.

### B.1.4 Pseudolabeling

For training the letter-based CTC models, we use the same Transformer-based encoder consisting of 270M parameters from [37] for the AM: the encoder of our acoustic models is composed of a convolutional frontend (1-D convolution with kernel-width 7 and stride 3 followed by GLU activation) followed by 36 4-heads Transformer blocks [38]. The self-attention dimension is 768 and the feed-forward network (FFN) dimension is 3072 in each Transformer block. We uses 64 $\times$ Nvidia 32GB V100 GPUs for training. The output of the encoder is followed by a linear layer to the output classes. We use dropout after the convolution layer. For all Transformer layers, we use dropout of 0.2 on the self-attention and on the FFN, and layer drop [12] value of 0.2. Token set for all acoustic models consists of 26 English alphabet letters, augmented with the apostrophe and a word boundary token. To speed up training, we also use mixed-precision training. We use the same SpecAugment and learning rate schemes as discussed above for STC models. We decay learning rate by a factor of 2 each time the WER reaches a plateau on the validation sets.

## B.2 Handwriting Recognition

For training the handwriting recognition models, we have used the open-source code[5] based on a prior work from [42] and adapted it for our use case. It uses depthwise separable convolutions as the main computational block and the model consists of about 39 million parameters. We use 8 $\times$ Nvidia V100 32GB GPUs for training the models.

It should also be noted that we were unable to find an open source implementation of Yousef et al. [43] and our baseline is our best attempt to reproduce their results.

We use Adam Optimizer [22] with an initial learning rate of 0.02 and run training using a batchsize of 8 per GPU for all the experiments. All image are scaled to a maximum width, height of 600, 32 pixels respectively before feeding to the neural network. We use exponential learning rate decay schedule with a gamma factor of $10^{-1/90000} \approx 0.99997$. We use random projection transformations as the augmentation method while training.

[5] `https://github.com/IntuitionMachines/OrigamiNet`

# C Appendix: A simple letter to word encoder

Since, we are working on direct-to-word models for ASR, a major hurdle would be to transcribe words not found in the training vocabulary. Recently, [8] propose an end-to-end model which outputs words directly, yet is still able to dynamically modify the lexicon with words not seen at training time. They use a convnet based letter to word embedding encoder which is jointly trained with the acoustic model and is able to accommodate new words at testing time.

In this work, we propose a simpler system for letter to word encoder which can be embedded into the acoustic model and thus avoiding the needs for a separate network. Consider $\mathcal{A}_L$, $\mathcal{A}_W$ are the alphabets for letters, words respectively and $l_{max}$ is the maximum length of a word. We assume $\mathcal{A}_L$ also contains two special letters - $c_{blank}$ which produces *blank* token required for CTC/STC models and $c_{pad}$ which is used to pad all words, *blank* token to the same maximum word length $l_{max}$. We let the acoustic model produce a $(|\mathcal{A}_L| \times l_{max})$ dimensional vector for each time frame.

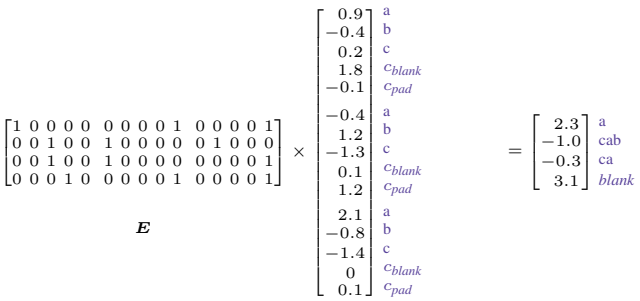



Figure 11: Letter to word encoder in action. Matrix $\boldsymbol{E}$ converts the letter scores over each timeframe to a score over words and *blank*. $\mathcal{A}_L = \{a, b, c, c_{blank}, c_{pad}\}$, $l_{max} = 3$, $\mathcal{A}_W = \{\text{a}, \text{cab}, \text{ca}\}$.

Since CTC/STC expect scores for $\mathcal{A}_W$ and *blank* for each time frame, we carefully construct a matrix $\boldsymbol{E}$ of 1s and 0s which converts a vector of size $(|\mathcal{A}_L| \times l_{max})$ to $(|\mathcal{A}_W| + 1)$ as shown in Figure 11 . Since we are using $c_{pad}$ token for padding, we can always assume every words and *blank* token is a sequence of $l_{max}$ letters. Each row in $\boldsymbol{E}$ is constructed by concatenating the one-hot representation of each letter (including $c_{pad}$) and it produces the score for a word or *blank*. We use $\mathcal{A}_L = \{a - z, ', c_{blank}, c_{pad}\}$ and $l_{max} = 20$ for all our experiments on LibriSpeech using words as the output tokens.

Table 3: WER comparison on LibriSpeech dev and test sets

| Method | Criterion | Model Stride/ Parameters | Output tokens | LM | Dev WER clean/other | Test WER clean/other |
|---|---|---|---|---|---|---|
| Transformer [8] | CTC | 8/∼ 300M | Words | - | 2.9/7.5 | 3.2/7.5 |
| | | | | Words 4-gram | 2.6/6.6 | 2.9/6.7 |
| | seq2seq | 8/∼ 300M | Words | - | 2.7/6.5 | 2.9/6.7 |
| | | | | word 4-gram | 2.5/6.0 | 3.0/6.3 |
| Transformer (Uses our simple letter to word encoder) | CTC | 8/70M | Words | - | 2.3/6.8 | 2.8/7.0 |
| | | | | Word 5-gram | 2.2/6.3 | 2.7/6.5 |

From Table 3, we can see that we are able to match the performance of the word based models from [8] using a much smaller acoustic model. Also, our system is simpler as we do not have a separate network for letter-to-word encoder.

# D Appendix: Hyperparameters for token insertion penalty, $\lambda$

In Table 4, we report the hyperparameters used for training STC models on LibriSpeech (Table 1) and IAM (Table 2). The best performing values are found by running a grid search over the hyperparameters and measuring performance on the development set. It can be seen that with higher $p\_drop$ values, a higher token insertion penalty values are usually preferred.

Table 4: Best performing hyperparameters used for token insertion penalty, $\lambda$ while training STC models for LibriSpeech (left) and IAM (right).

| WEAK LABEL GEN. STRATEGY | $p_0$ | $p_{max}$ | $t_{1/2}$ |
|---|---|---|---|
| P_DROP=0.1 | 0.1 | 0.3 | 8000 |
| P_DROP=0.4 | 0.4 | 0.7 | 8000 |
| P_DROP=0.7 | 0.5 | 0.9 | 8000 |
| SPLIT ALL SAMPLES INTO 3 PARTS RANDOMLY; ASSIGN P_DROP=0.1,0.4,0.7 FOR THE SPLITS | 0.3 | 0.6 | 8000 |
| SPLIT ALL WORDS INTO 3 PARTS RANDOMLY; ASSIGN P_DROP=0.1,0.4,0.7 FOR THE SPLITS | 0.5 | 0.7 | 8000 |

| P_DROP | $p_0$ | $p_{max}$ | $t_{1/2}$ |
|---|---|---|---|
| 0.1 | 0.5 | 0.8 | 10000 |
| 0.3 | 0.5 | 0.9 | 10000 |
| 0.5 | 0.5 | 0.9 | 10000 |
| 0.7 | 0.7 | 0.9 | 10000 |

### D.1 Sensitivity of the model performance when setting token insertion penalty, $\lambda$

In Table 5, we report the effect of hyper parameter choices on the model performance for IAM dataset for $p\_drop = 0.1$ and $p\_drop = 0.7$. We see that the model performance is more sensitive with respect to hyperparameter choices for $p\_drop = 0.7$ compared to $p\_drop = 0.1$.

Table 5: Effect of hyper parameter choices on the model performance for IAM dataset. $p\_drop = 0.1$ (left) , $p\_drop = 0.7$ (right).

$p\_drop = 0.1$

| $p_0$ | $p_{max}$ | $t_{1/2}$ | DEV CER |
|---|---|---|---|
| 0.5 | 0.8 | 10000 | 5.0 |
| 0.3 | 0.6 | 10000 | 5.0 |
| 0.5 | 0.9 | 10000 | 5.0 |
| 0.4 | 0.8 | 10000 | 5.1 |
| 0.6 | 0.8 | 10000 | 5.1 |
| 0.5 | 0.8 | 5000 | 5.2 |
| 0.2 | 0.5 | 10000 | 5.3 |
| 0.2 | 0.4 | 10000 | 5.3 |
| 0.2 | 0.4 | 30000 | 5.4 |
| 0.2 | 0.6 | 10000 | 5.4 |
| 0.1 | 0.3 | 10000 | 5.6 |
| 0.2 | 0.4 | 50000 | 5.6 |

$p\_drop = 0.7$

| $p_0$ | $p_{max}$ | $t_{1/2}$ | DEV CER |
|---|---|---|---|
| 0.7 | 0.9 | 10000 | 22.7 |
| 0.65 | 0.9 | 10000 | 23.6 |
| 0.65 | 0.95 | 10000 | 24.9 |
| 0.6 | 0.95 | 10000 | 26.1 |
| 0.5 | 0.9 | 10000 | 27.7 |
| 0.8 | 0.9 | 10000 | 30.3 |
| 0.5 | 0.95 | 10000 | 30.5 |
| 0.7 | 0.8 | 50000 | 32.9 |
| 0.6 | 0.9 | 10000 | 33.9 |
| 0.7 | 0.8 | 10000 | 34.6 |
| 0.7 | 0.95 | 10000 | 35.9 |
| 0.3 | 0.9 | 10000 | 41.0 |
| 0.5 | 0.9 | 10000 | 42.7 |
| 0.7 | 0.75 | 10000 | 46.7 |
| 0.8 | 0.95 | 10000 | 48.3 |