Transactions on Machine Learning and Artificial Intelligence - Vol. 10, No. 6
Publication Date: December, 25, 2022
DOI:10.14738/tmlai.106.13515.
Gal, O., Michaeli, R., & Doytsher, Y. (2022). Self-Supervised Learning in Hebrew – Model to Practice Framework. Transactions on Machine Learning and Artificial Intelligence, 10(6). 35-51.

Self-Supervised Learning in Hebrew – Model to Practice Framework

Oren Gal
Israel Institute of Technology, Technion, Israel

Rafi Michaeli
Tel Aviv University, Israel

Yerach Doytsher
Israel Institute of Technology, Technion, Israel

ABSTRACT

In this paper, we present the current state-of-the-art models for Automatic Speech Recognition due to a self-supervised training implemented on Hebrew language. The motivation behind using self-supervised learning is that even though we wouldn't probably get the accuracy rates as if we choose a supervised learning, we still can achieve amazing results with relatively low amount of data. This way of training allows us to train a model on unlabeled data (or to use a pre-trained model, which is always more accessible. It's goal in the first unsupervised phase is to learn some good representations from raw audio samples, which can be useful for speech recognition tasks, without using any label data. Then, the model can be fine-tuned on a particular dataset for a specific purpose. It means that our involvement really occurs in the last layers of the model. This kind of training proved to be very powerful. We present complete framework from model to practice with simulations and training model and present an impressive result on Hebrew.

Keywords: CNN, self-supervised, N-gram, Natural Language Processing.

INTRODUCTION AND BASICS

Modern neural networks have been highly successful in a wide variety of applications ranging from Computer Vision [3] to Natural Language Processing [1]. However, these successes have come on the back of training large models on massive, labeled data sets, which may be costly or even infeasible to obtain in other applications. For instance, applying deep networks to the task of cancer detection requires medical images that can only be labeled with the expertise of healthcare professionals, and a single accurate annotation may come at the cost of a biopsy on a patient [2].

First, we define some basic fields of model training.
Semi-Supervised learning – The ability to train a model based on big amount of unlabeled data, train with that only the first layers of our model and then fine tune the model using our own custom labeled data.

Supervised Learning – kind of model training that based on a lot of labeled data. This method requires a very quality data as well as big amounts of data, but reaches most of the times, the best results compare to other learning methods.

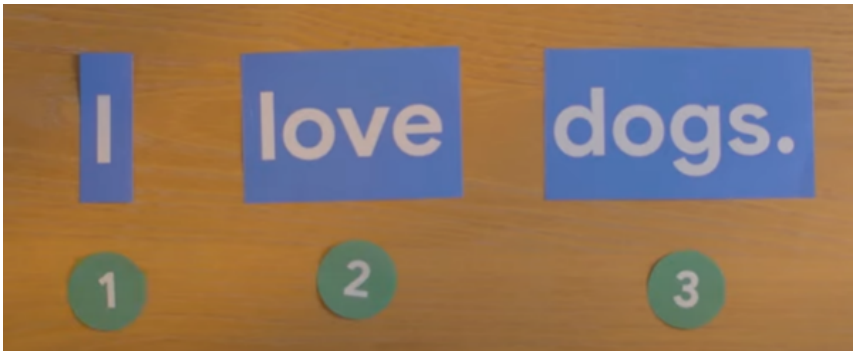
Transformer – One of the most common methods in the domain of Deep learning. A type of neural network that takes a data of a certain type and turns it into another different type of data. Transformers are models that can translate a text, so it is a crucial tool in the NLP domain. Before Transformers are came up, the way we used in deep learning for understanding text was by using RNN (Recurrent Neural Network).

Cons of RNN:

- RNN takes a full sentence as input but process a word one at a time.
- The keyword in NLP and text is sequential, and the order of the word is what matters. RNN never really did well at handling a large sequence of text, like a long paragraph.
- By the time they were analyzing the end of the paragraph, they had forgotten what happened in the beginning.
- RNN were hard to train. Since they process the word sequentially, they couldn't parallelize well means that we can't speed up the training process by throwing up lots of GPU's at them.

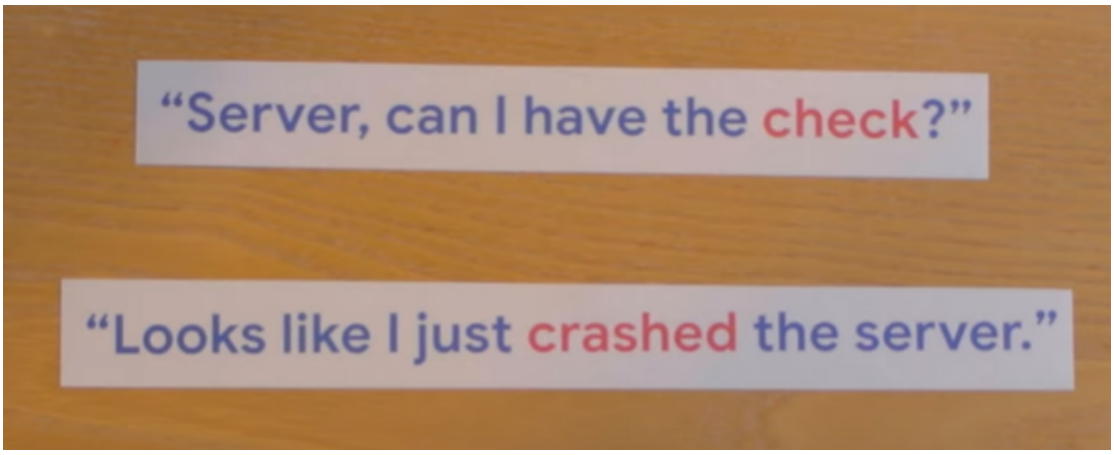
How does a transformer work?

- Positional encoding – Instead of looking at word sequentially, we take each word in the sentence, and before feeding it to the neural network we slap a number to it.



In that way we store the information about the order of the words in the data itself rather than in the structure of the network. Then, when we train the model with large amount of data, it learns how to interpret those positional encoding. In this way, the neural network learns the importance of word order from the data:

- **Attention** – The attention mechanism is neural network that allows a text model to look at every single word in the original sentence when deciding about how to transform the input word to the output word in the output sentence (useful especially in translation). The model knows which words it should attend to by learning it from the data over time.
- **Self Attention –** The real innovation of transformers. This is a twist of traditional attention. The real incredible thing about neural networks is that as they analyze tons of text data, they begin to build up an internal representation or understanding the language automatically. They learn the underlying secrets of the language as synonymous words, grammar rules, gender, tense and so on.



Self attention allows a neural network to understand a word in the context of the words around it. So when the model looks on the word "Server" from the first sentence from the above example, he might attend to the word "check" to disambiguate from a human server versus a mail one. In the second sentence, the model might attend to the word "crashed" to determine that the server is a machine. It also helps the neural network to disambiguate words, recognize parts of speech and identify word tense.

Bert – Google architecture that implement a transformer. When people say the name "Bert" they usually mean to the Google's language model.
Phoneme - the smallest possible unit of sound in a particular language, usually represented by one or two letters.

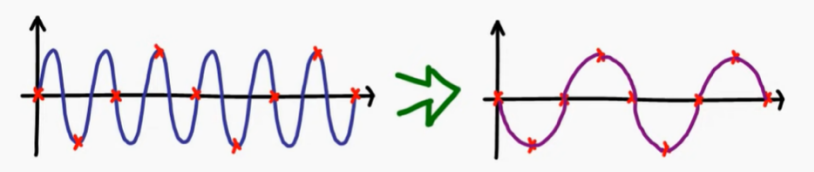
Speech to text full pipeline
The PCM phenomena
We record a raw data, which is a human voice and sample it with a sample rate of 16Khz and with a PCM as an encoding method.

The PCM phenomena, includes recording microphone does is convert the air pressure into electrical signal. Our method is a linear one, which means that the number described by two bytes in the output will be relative to the air pressure fed into the microphone. The higher this number is, the higher is the pressure fed into the microphone. For *wav2vec* models, working with nonlinear methods is not recommended, therefore we work with PCM.

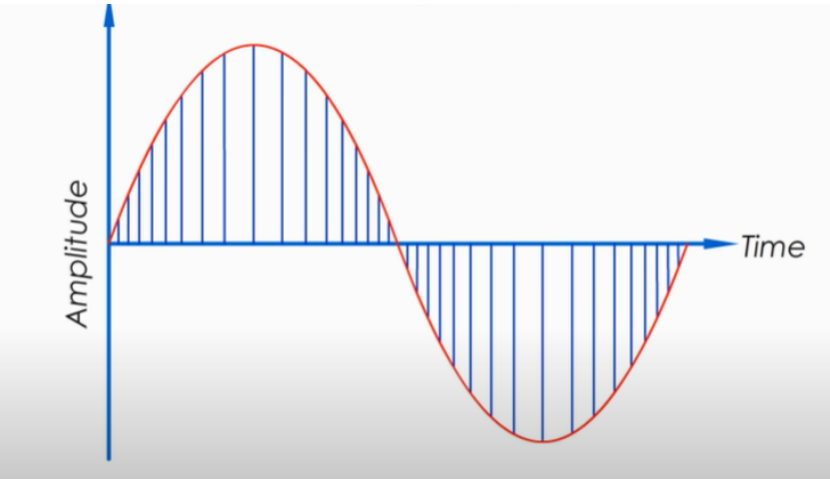
PCM consists of three steps:
Sampling – is the process of finding the sufficient number of samples so that the original signal can be represented by those samples completely and it should be possible to reconstruct the original signal. The human speaking frequencies range is between 100Hz-4Khz so we chose the sampling rate to be 16Khz because of the Nyquist-Shannon sampling theorem which says that perfect reconstruction is guaranteed for a sample rate bigger or equivalent to $2 * (f_{signal})$.

$$\boldsymbol{sample\ rate} < \mathbf{2} * \boldsymbol{f_{signal}} \quad (1)$$

URL: http://dx.doi.org/10.14738/tmlai.106.13515



Therefore the constraints is:



$$\boldsymbol{sample\ rate} \geq \mathbf{2} * \boldsymbol{f_{signal}} \qquad (2)$$

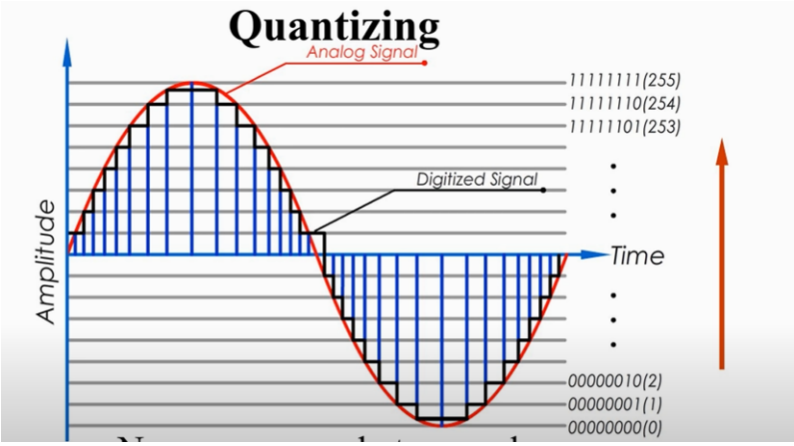
For the maximum human frequency of 4Khz we need a sample rate of at least 8Khz. The problem with taking 8Khz (which is exactly double) as the sample rate is that while dealing with similar letters (Hebrew letters with high phonetic similarity like: 'א','ע','ה') there is a big difficulty to distinguish between them if we lose the high frequencies. Accordingly, we chose the 16Khz just to be sure that we will cover 99% from the human spectrum.

Quantizing – The process of fitting each sample a specific output value. Sampling converts a time varying signal into a discrete time signal of real numbers. Real numbers are any number that can be found on a line segment. However, we only need a finite set of values which can be converted into binary numbers. This is the process called quantizing [5].

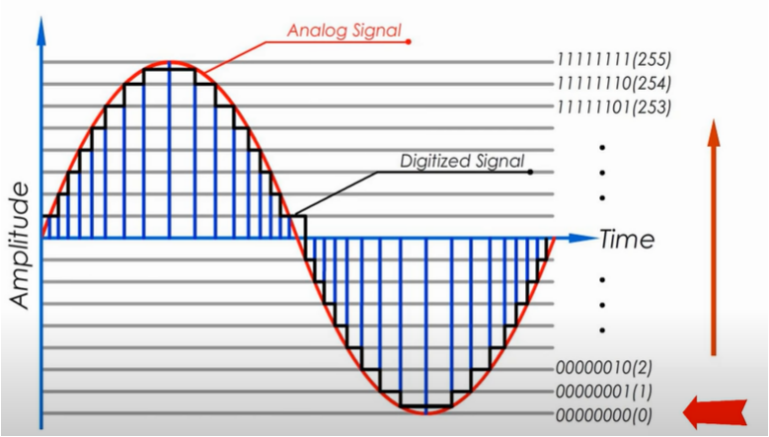
For make this process properly, we need to define the bit depth. Bit depth is the number of bits in each sample (The most common are 8,16,24). The higher the bit depth is, the more options for voltage representations we have.

$$\begin{gathered} \mathbf{8-bit\ depth}:\ 2^8 = 256 \\ \mathbf{16-bit\ depth}:\ 2^{16} = 65{,}536 \\ \mathbf{24-bit\ depth}:\ 2^{24} = 16{,}777{,}216 \end{gathered} \qquad (3)$$

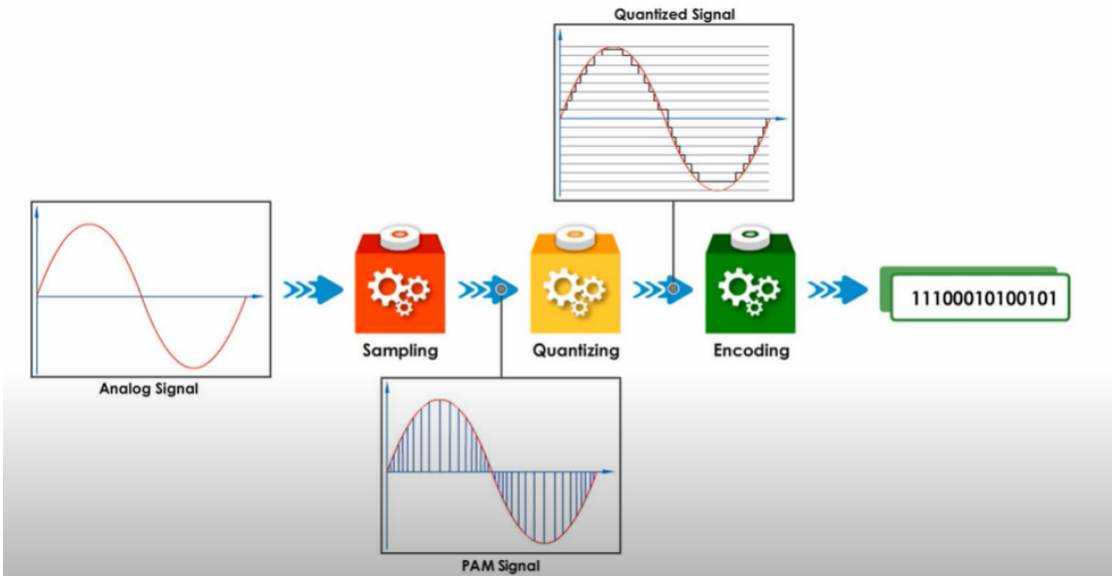
Each level is associated with a specific bit value. We can summarize and say that quantizing is the process of rounding off the amplitude of flat top samples to a manageable number of levels. We used the 8-bit depth:



1. **Encoding -** Now we can encode the samples with the number of bits we choose, in our case 8 bits. The highest level will be 11111111 and the lowest level will be 00000000. Everything in between will be represented by 8 bits as well.



We can summarize the full analog to digital conversion as described in the chart below:



URL: http://dx.doi.org/10.14738/tmlai.106.13515

In the end of this process, each analog signal is being sampled with sample rate of 16Khz and a linear encoding method called PCM create a wav file represent our audio. Each wav file is a 1-D vector of numbers. These wav files sent to the next level in the process – the wav2vec model.

We take the sampled data (wav files) and feed the wav2vec model with it in order to split the signal into a sequence of characters. Let's talk in detail about the model and its training.

Wav2Vec 2.0 and N-gram - Wav2Vec 2.0:

The **Wav2Vec** 2.0 is one of the current state-of-the-art models for Automatic Speech Recognition due to a self-supervised training. The motivation behind using self-supervised learning is that even though we wouldn't probably get the accuracy rates as if we choose a supervised learning, we still can achieve amazing results with relatively low amount of data.

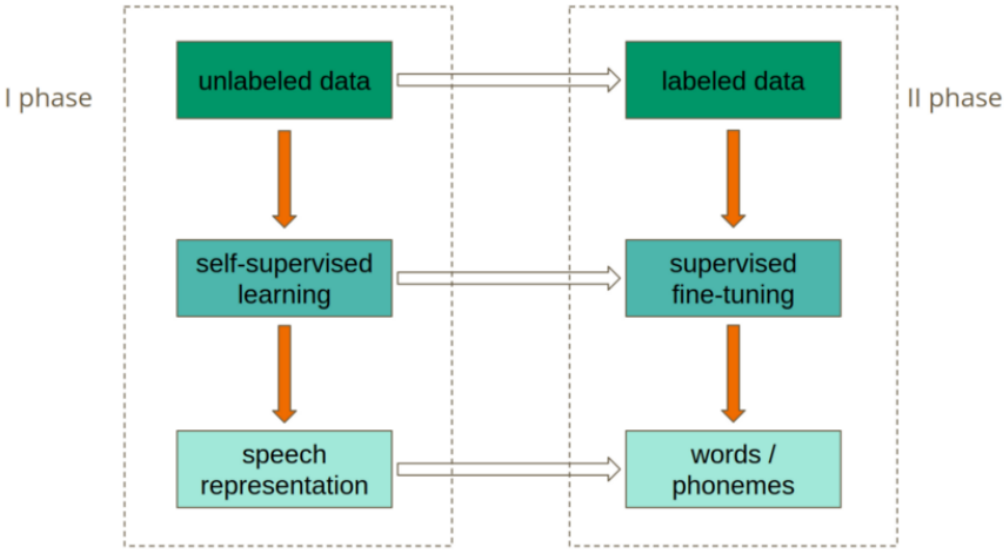
This way of training allows us to train a model on unlabeled data (or to use a pre-trained model, as we did) which is always more accessible. Its goal in the first unsupervised phase is to learn some good representations from raw audio samples, which can be useful for speech recognition tasks, without using any label data. Then, the model can be fine-tuned on a particular dataset for a specific purpose. It means that our involvement really occurs in the last layers of the model. This kind of training proved to be very powerful.

To sum up, the model is trained in two phases.

- The first phase is in a self-supervised mode, which is done using unlabeled data and it aims to achieve the best speech representation possible.

This phase is the main advantage of the model. Learning a very good speech representation enables achieving state-of-the-art results on a small amount of labeled data.

- The second phase of training is supervised fine-tuning, during which labeled data is used to teach the model to predict particular words or phonemes.



During the first phase of training, in the self-supervised mode, the model is trained without the linear projection generating an output prediction. The main idea of pre-training is like BERT: part of the transformer's input is masked, and the aim is to guess the masked latent feature vector representation. The model has been improved due to **contrastive learning [4]**.

Contrastive learning is a concept in which the input is transformed in two different ways. Afterwards, the model is trained to recognize whether two transformations of the input are still the same object. In Wav2Vec 2.0, the transformer layers are the first way of transformation, the second one is made by quantization. for a masked latent representation Z_t, we would like to get such a context representation C_t to be able to guess the correct quantized representation q_t among other quantized representations.

N-Gram

Given a sequence of N-1 words, an N-gram model predicts the most probable word that might follow this sequence. It's a probabilistic model that's trained on a corpus of text. Such a model is useful in many NLP applications including speech recognition, machine translation and predictive text input. An N-gram model is built by counting how often word sequences occur in corpus text and then estimating the probabilities. An N-gram model is one type of a Language Model (LM), which is about finding the probability distribution over word sequences. In speech recognition, input may be noisy, and this can lead to wrong speech-to-text conversions. N-gram models can correct this based on their knowledge of the probabilities. Likewise, N-gram models are used in machine translation to produce more natural sentences in the target language. N-Grams begins with the task of computing $p(\frac{w}{h})$, the probability of a word w given some history h.

Let's say we have the history ,h, "I need to eat" and we want to calculate the probability of the word, w, "chocolate". One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see "I need to eat", and count the number of times this is followed by "chocolate". This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$p(\text{choclate/I need to eat}) = \frac{C(I\ need\ to\ eat\ choclate)}{C(I\ need\ to\ eat)} \quad (4)$$

With a large enough corpus, we can compute these counts and estimate the probability from this equation. While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. For this reason, we'll need to introduce more clever ways of estimating the probability of a word w given a history h of an entire word sequence.

We'll represent a sequence of N words either as $W_1 \dots W_n$ or $W_{1:n}$. It means that the expression $W_{1:n-1}$ means the string $W_1, W_2, \dots, W_{n-1}$. For the joint probability of each word in a sequence having a particular value, $P(X_1 = w_1, X_2 = w_2, X_3 = w_3, \dots, X_n = w_n)$ let's write $P(w_1, w_2 \dots, w_n)$ and the probability of the entire sequence is defined by the chain rule of the probabilty:

$$P(X_1, \dots, X_n) = P(X_1)P(X_2/X_1)P(X_3/X_{1:2}) \dots P(X_n/X_{1:n-1}) = \prod_{k=1}^{n} P(X_k/X_{1:k-1}) \quad (5)$$

Apply this rule chain on our words to get:

URL: http://dx.doi.org/10.14738/tmlai.106.13515

$$P(w_{1:n}) = P(w_1)P(w_2/w_1)P(w_3/w_{1:2})..P(w_n/w_{1:n-1}) = \prod_{k=1}^{n} P(w_k/w_{1:k-1}) \quad (6)$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. The main idea is that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n/w_{1:n-1})$.

we can't just estimate by counting the number of times every word occurs following every long string, because language is creative and any particular context might have never occurred before! The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words.

Let's make it more clear by using an example. The bigram model, for example, approximates the probability of a word given all the previous words $P(w_n/w_{1:n-1})$ by using only the conditional probability of the preceding word $P(w_n/w_{n-1})$, which is a approximation. In other words, , instead of computing the probability p(choclate/I need to eat), we approximate it with the probability p(choclate/eat). When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n/w_{1:n-1}) \approx P(w_n/w_{n-1}) \quad (7)$$

The assumption that the probability of a word depends only on the previous word is Markov called a Markov assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) n-gram to the trigram (which looks two words into the past) and thus to the n-gram (which looks n−1 words into the past). Let's see a general equation for this n-gram approximation to the conditional probability of the next word in a sequence. We'll use N here to mean the n-gram size, so N = 2 means bigrams and N = 3 means trigrams. Then we approximate the probability of a word given its entire context as follows:

$$P(w_n/w_{1:n-1}) \approx P(w_n/w_{n-N+1:n-1}) \quad (8)$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k/w_{k-1}) \quad (9)$$

How do we estimate these bigrams or n-gram probabilities? An intuitive way to estimate probabilities is called maximum likelihood estimation or MLE. We get maximum likelihood estimation the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.1 normalize For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the sum of all the bigrams that share the same first word w_{n-1}.

$$P(w_n/w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \quad (10)$$

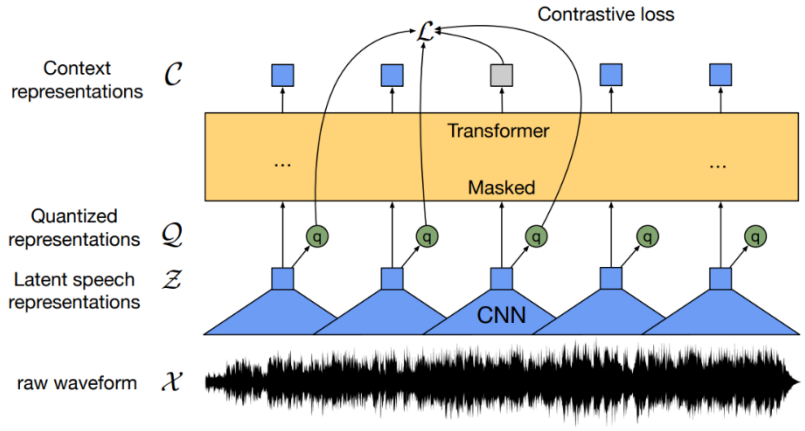
We can simplify this equation, since the sum of all bigram counts that start with a given wordw_{n-1}must be equal to the unigram count for that word w_{n-1} (the reader should take a moment to be convinced of this):

$$P(w_n/w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \quad (11)$$

In our pipeline we found the bigrams probabilities last equation, but also trigrams and four grams as well to help the computer to consider the content of each given predicted characters sequence [6].

Wav2Vec 2.0 - General overview Model Architecture and Training process

In this training phase, our model gets as an input short chunks of wav files (20 millisecond long for each one) and its goal is to recognize the accurate sound heard in this chunk. The decision of dividing the chunks to 20ms derived from the fact that each phoneme is about 40-50 ms.



Our architecture consists of three main parts:

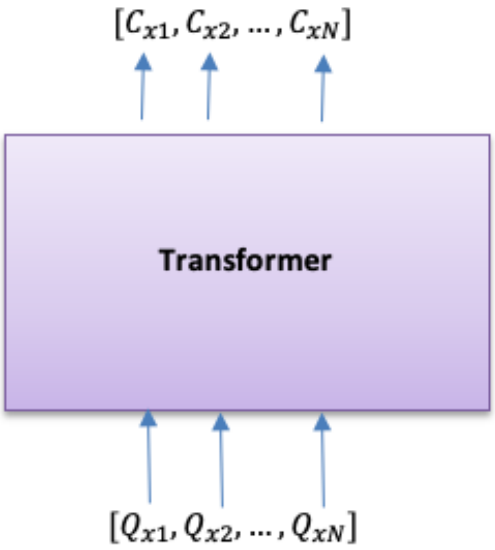
1. convolutional layers that process the raw waveform input to get latent representation.
2. transformer layers, creating contextualized representation.
3. linear projection to output.

The wav2vec 2.0 model follow the design choices described in Baevski et al. (2020c). As we said in the last section, we feed this model with wav files which being converted, or more precisely being vectorize by its features, during the process to 1-D vectors of numbers. It means that X which is the original waveform, got mapped to an analog vector of features, $X \rightarrow Z$.

These vectors go through around 7 layers of CNN which learn their representation. Z is basically a vector followed by the structure $[Z_1, Z_2, \dots, Z_T]$ and represents the outputs latent speech representations. Then the quantization is got into the process. For the purpose of training the model, feature encoder representations are discretized to $[Q_1, Q_2, \dots, Q_T]$ with a quantization module that maps $Z \rightarrow Q$ eventually, to represent the targets in the self-supervised learning objective. After the conversion to Q sequence, the model is using a transformer to learn the contextual representation. The transformer is a neural network by itself, and we need to train the network as well. So how can we train the transformer? The

URL: http://dx.doi.org/10.14738/tmlai.106.13515

transformers use something called masking. The masking is an operation done on some vectors – The found latent speech representations are fed into a transformer which makes $Q \rightarrow C$ in order to build representations that capturing information from the overall sequence - $[C_1, C_2, \dots, C_T]$. The transformer makes some of the input vectors to be 0 for example and hides part of the data. The idea of this concept is to predict these masked vectors in the output of the transformer and by doing it enforce the model to be more general. Each C_x should be predicted to be equal to its appropriate element from Q_x sequence.



We can refer to each element in this Q vector, as a code. Then the model use a "code book" to map each Q element into C element. The sequent can be described in the following table:

Table 1. Q element into C element

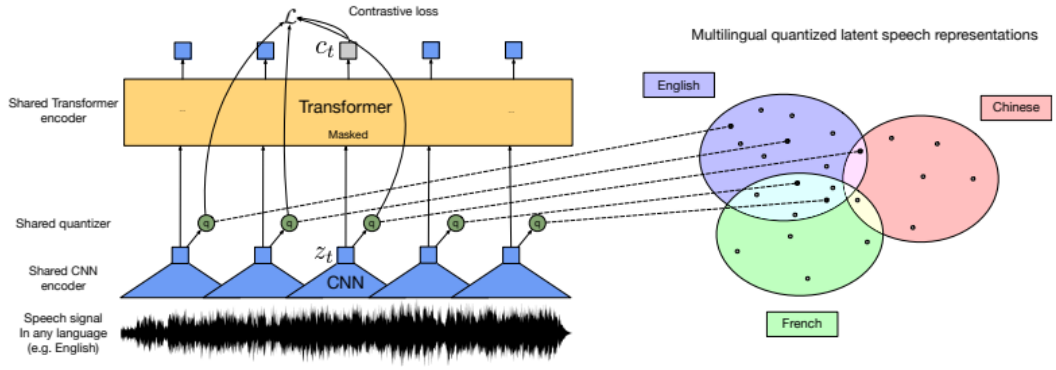
Q Sequence	***C Sequence***
$[Q_{x1}, Q_{x2}, \dots, Q_{xN}]$.	$[C_{x1}, C_{x2}, \dots, C_{xN}]$.
$[Q_{y1}, Q_{y2}, \dots, Q_{yN}]$.	$[C_{y1}, C_{y2}, \dots, C_{yN}]$.
$[Q_{z1}, Q_{z2}, \dots, Q_{zN}]$.	$[C_{z1}, C_{z2}, \dots, C_{zN}]$.
$[Q_{k1}, Q_{k2}, \dots, Q_{kN}]$.	$[C_{k1}, C_{k2}, \dots, C_{kN}]$.

To complete the learning process properly, we must use some loss function. In our model we use the contrast loss – This function keeps only the predicted values that will be close enough to the input values and throw away the predicted values that far from our input. It means that if $C_{x1} \approx Q_{x1}$ the function would keep Q_{x1}. Eventually, after the model supervised training we can trust the wav2vec model's output to be our spoken characters. Now let's talk about the "Bert" part. We want the output characters to also be taken in a logical content way. For this, we need to use a language model or a model that takes into account the importance of the words order. And for that, we use the N-gram model which will be described later.

Our pipeline

In our pipeline, we took an already pre-trained model to the unsupervised training phase and then make a fine tuning on it. We chose to use the XLSR as our pre-trained model that Facebook has trained on raw waveform of speech in multiple languages. The large number of languages

makes this model to be versatile and adaptive to most of the languages in the world. That's why even though this model has not been trained on Hebrew specifically, the fact that it has been trained on variety languages such as: Russian, Arabic, English that some of them are very similar to Hebrew, makes it a good choice in understanding the structure of the Hebrew language as well. So, the XLSR is the model that the wav2vec is based on in the phase of the unsupervised training. We make this step to teach the model the ability of understanding a wav file correctly as well as the structure of the Hebrew language. In this unsupervised training step the Wav2Vec model builds its first layers. We can say that the first layers of wav2vec model know how to understand a language from listening due to XLSR model. We will talk in detail about the architecture of XLSR and wav2vec in the training section.



In order to do a fine-tuning, we split our wav files into 20ms chunks, and from each chunk our goal is to extract a single Hebrew character or a sequence via the wav2vec model.

Wav2Vec 2.0 different phases + Training

Labeled DATA Organizing and fine-tuning process

For the making the model appropriate to our needs, we must train its last layers. This part is the supervised part, and its main part is to do a fine tuning. Training only the last layers of this model makes the whole process much cheaper and faster. The fine-tuning process starts by creating a bunch of sound waves samples and their appropriate label (e.g. bunch of sound waves and their exact transcription). The process of creating the labels must be done by experienced transcribers. Every mistake at this phase between the label and the wave file, might has a massive bad undesired change in the model weights. Since that, we used an approximately 20 hours of data, clean them and made subtitles (using experienced transcribers) to each sample. The subtitles must be insanely accurate – not only in the speaking content but also in the accurate time that the speak exactly heard in the wav file. Another important step is required in this phase which is building a character's dictionary. As we said, in the unsupervised training the model doesn't really learn a language. The language learning is done in this phase, and we must tell the model which characters we expect from him to predict. Our dictionary composed of all the 22 Hebrew letters, space character (space between words) and two special characters: unknown character and padding character (This character will be described deeply in the CTC phase). The reason we give an unknown character is because sometimes during the training phase, we as humans meet complex situations don't allow us to recognize characters properly (especially from acoustic reasons). In these situations, we better tell the model to put the unknown character rather than to guess a random

URL: http://dx.doi.org/10.14738/tmlai.106.13515

character and as a result harm the model weights during the training. Then, when our model meets such complex situations, he would alert about it.

Labeled DATA Preprocessing

In this phase we are fully focus on cleaning our Data. In this domain, the most important action is cleaning the data, since this action leads to the most meaningful impact on the model evaluation results. Therefore, we made a small filtering due to available resources as well as improving our model understanding during the training phase. From performance point of view, our motivation is removing the extremely short samples, such as 1-2 seconds samples. The reason we want to remove such samples is the fact that in such samples our model will not hear any logical sound pronunciation and such samples only gets our model to be confused. Punctuations have been removed as well since they add a noise to the model (Punctuations are not physically heard in the wave file, and non-heard things confuse the models most of the times – therefore we removed them from the labeled data). Another step in this phase is converting each of the Hebrew suffix letters ('ם','ך','ף','ץ','ן') to the regular version of the letter. At the end of the process, each suffix letter will be transformed back to its origin so the transcription will be exactly as it should be. Also, words that pronounced differently than their spelling, such as digits, will be removed. From resources point of view, it is important to know that these kinds of models, tend to consume a lot of RAM. The RAM consuming is increasing directly by the size of the (maximum sample size)^2 . It means that if we have a 5 second sample and a 7 second sample (Let's assume that these are the only samples in our dataset), so the RAM consuming has doubled - $7^2 \sim \mathbf{2} * 5^2$, which increases the RAM value by dozens of giga RAM. Due to this fact, we made some additional filter – Too long samples (samples longer than 10 seconds) have been removed from our dataset. After we cleaned our data, we are ready to feed the model with these samples.

Wav2Vec 2.0 – Feature Encoder

The first level in the process is the feature extraction. Computers don't really know how to work with raw textual data. Therefore, we need to extract the wave chunks features and encode them to reconstruct the exact characters by decoding the features at the end of the process. The encoder consists of several blocks containing a temporal convolution followed by layer normalization to every channel and a GELU activation function and the raw waveform input to the encoder is normalized to zero mean and unit variance. The output of this step , Z is a matrix that each column in it represent a features chunk.

Wav2Vec 2.0 – Quantization Module

For self-supervised training we discretize the output of the feature encoder Z to a finite set of speech representations via product quantization. Product quantization amounts to choosing quantized representations from multiple codebooks and concatenating them. Given G codebooks, or groups, with V entries $e \in R^{V \times d/G}$, we choose one entry from each codebook and concatenate the resulting vectors $e_1, \ldots, e_G$ and apply a linear transformation $\mathrm{R}^{\mathrm{d}} \rightarrow \mathrm{R}^{\mathrm{f}}$ to obtain $\mathrm{Q} \in \mathrm{R}^{\mathrm{f}}$ [7].

For the purpose of training the model, feature encoder representations are discretized to $[Q_1, Q_2, \ldots, Q_N]$ .with a quantization module $Z \rightarrow Q$ to represent the targets in the self-supervised learning objective. The quantization is based on product quantization (Jegou et al.,

2011; Baevski et al., 2020b) by choosing quantized representations from $G = 2$ codebooks with $V = 320$ entries each. The result is concatenated to obtain Q elements.

The Gumbel softmax enables choosing discrete codebook entries in a fully differentiable way. We use the straight-through estimator and setup G hard Gumbel softmax operations. The feature encoder output Z is mapped to $l \in R^{G \times V}$ logits and the probabilities for choosing the v-th codebook entry for group g are:

$$P_{g,v} = \frac{\exp(l_{g,v}+n_v)/\tau}{\sum_{k=1}^{V}\exp(l_{g,k}+n_k)/\tau} \quad (12)$$

where τ is a non-negative temperature, $n = -log(-log(u))$ and u are uniform samples from $u(0,1)$. During the forward pass, codeword i is chosen by $i = argmax_j P_{g,j}$ and in the backward pass, the true gradient of the Gumbel softmax outputs is used [8].

Training the model

To pre-train the model we mask a certain proportion of time steps in the latent feature encoder space, like masked language modeling in BERT. The training objective requires identifying the correct quantized latent audio representation in a set of distractors for each masked time step and the final model is fine-tuned on the labeled data.

Masking

We mask a proportion of the feature encoder outputs, or time steps before feeding them to the context network and replace them with a trained feature vector shared between all masked time steps. we do not mask inputs to the quantization module. To mask the latent speech representations output by the encoder, we randomly sample without replacement a certain proportion p of all time steps to be starting indices and then mask the subsequent M consecutive time steps from every sampled index.

During pre-training, we learn representations of speech audio by solving a contrastive task L_m which requires to identify the true quantized latent speech representation for a masked time step within a set of distractors. This is augmented by a codebook diversity loss L_d to encourage the model to use the codebook entries equally often. $L = L_m + \alpha L_d$ where α is a tuned hyperparameter.

Contrastive Loss

Given context network output C_t centered over masked time step t, the model needs to identify the true quantized latent speech representation q_t in a set of $K + 1$ quantized candidate representations $\tilde{q} \in Q_t$ which includes q_t and K distractors. Distractors are uniformly sampled from other masked time steps of the same utterance.

The loss is defined as:

$$L_m = -log\frac{\exp(sim(C_t, q_t)/k)}{\sum_{\tilde{q}\sim Q_t}\exp(sim(C_t,\tilde{q})/k)} + \alpha L_d \quad (13)$$

where we compute the cosine similarity $sim(a,b) = a^T b/||a||||b||$ between context representations and quantized latent speech representations.

URL: http://dx.doi.org/10.14738/tmlai.106.13515

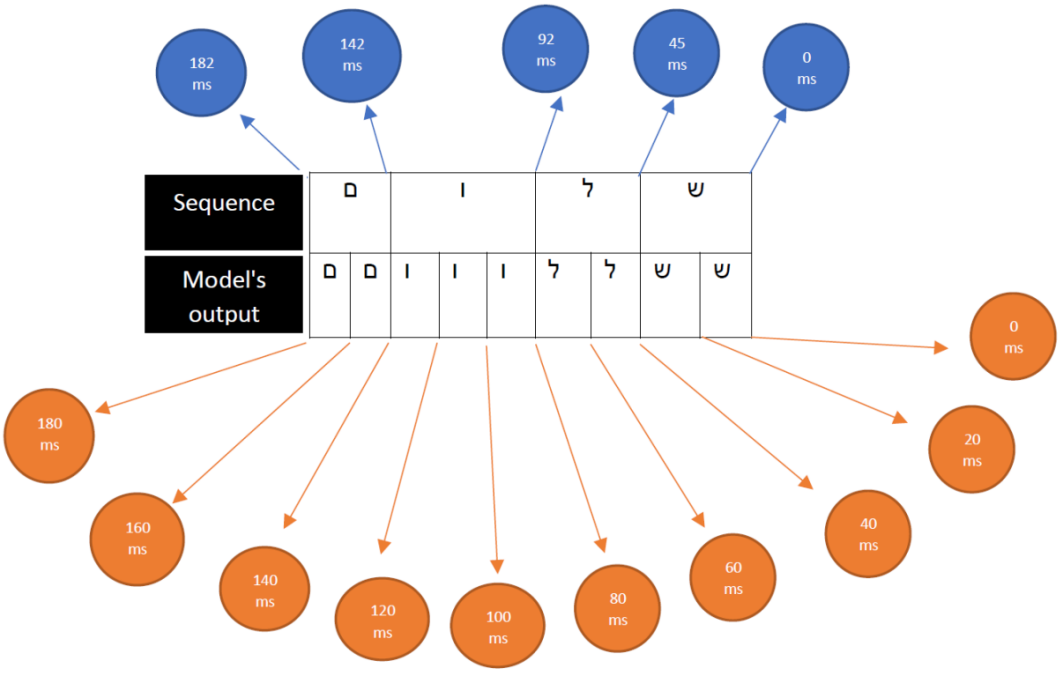
Wav2Vec 2.0 – Contextualized representations with transformers

We used the transformer architecture to learn the contextualized representations. This context network architecture follows BERT. The transformer takes as input the features vectors that has been extracted from the feature encoder level. Instead of fixed positional embeddings which encode absolute positional information, we use a convolutional layer which acts as relative positional embedding. Finally, we add the output of the convolution followed by a GELU to the inputs and then apply layer normalization.

Another extra process has done in this phase, called Connectionist Temporal Classification (CTC).

Connectionist Temporal Classification (Features extracting)

One of the main problems with this chunk length value of 20ms, is that sometimes the model gives back as a result, the same letters that he heard in the former chunk and returns a non-logical sequence. For example:



So the model automatically remove the same letters. But what if I want to spell the word "לחמם" or "Hello"?

If the model automatically removes the similar adjacent letters, it won't be able to spell many words such hello. So, this model has a special character called "padding character" – which means that when the model recognize this situation, it will add a padding character between similar adjacent letters and that's how it knows how to don't remove it. At the end of the process the model will remove the padding characters by itself, so these characters are not appeared in the final output.

To sum up, Wav2Vec2 is fine-tuned using Connectionist Temporal Classification (CTC), which is an algorithm that is used to train neural networks for sequence-to-sequence problems and

mainly in Automatic Speech Recognition and handwriting recognition. After we got the exact characters.

N-Gram model

The output of the Wav2Vec 2.0 model feeds another model named N-gram. The CTC does good job in recognizing context – he recognizes some tokens before the current token and some after the current token, but it is too narrow context recognizing and most of the time we need a tool that will know how to recognize a sequence of characters (e.g. words) by the content better than the CTC. Therefore, we have the N-gram model right after the wav2vec 2.0 model. This model looks on a logical sequence of characters – e.g. words that appears in a Hebrew language dictionary (or any other language of course) or a logical sequence of words. We train the N-gram model and chose it because this model gives a score to each word in a dictionary and gives a score to each sequence. The score represents how much the word is common in the language. We trained the model on a big corpus of text took from the internet. The text file that has written, composed of many text files concatenated to each other and we calculated the Bi-grams, trigrams, forth-grams scores by the equations:

$$P(w_n/w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \quad (14)$$

$$P(w_n/w_{n-2}w_{n-1}) = \frac{C(w_{n-2}w_{n-1}w_n)}{C(w_{n-2}w_{n-1})} \quad (15)$$

$$P(w_n/w_{n-3}w_{n-2}w_{n-1}) = \frac{C(w_{n-3}w_{n-2}w_{n-1}w_n)}{C(w_{n-3}w_{n-2}w_{n-1})} \quad (16)$$

MODEL EVALUATION AND RESULTS

The most important parameters when training such a model are: WER (Word Error Rate) and CER(Character Error Rate). WER checks if the model output word is correct relatively to the labeled word and the CER checks if the model output character is correct relatively to the labeled character. In the CTC phase, while the model creates characters it also tries to reduce the CER in parallel. The WER is derived from the CER since once a word has even one mistaken character, the whole word is declared as misspelled word. Due to this fact, The WER is bigger than the CER (in approximately 3 times). In addition to these parameters, we evaluate the Contrastive Loss. The loss is a metric that defines how far our model results than the reality results, of course we want our model loss to be 0. The learning rate is changed by Adam optimizer. After the model results have been given, we plotted a confusion matrix that indicates directly about the letters switches, insertions and omissions.

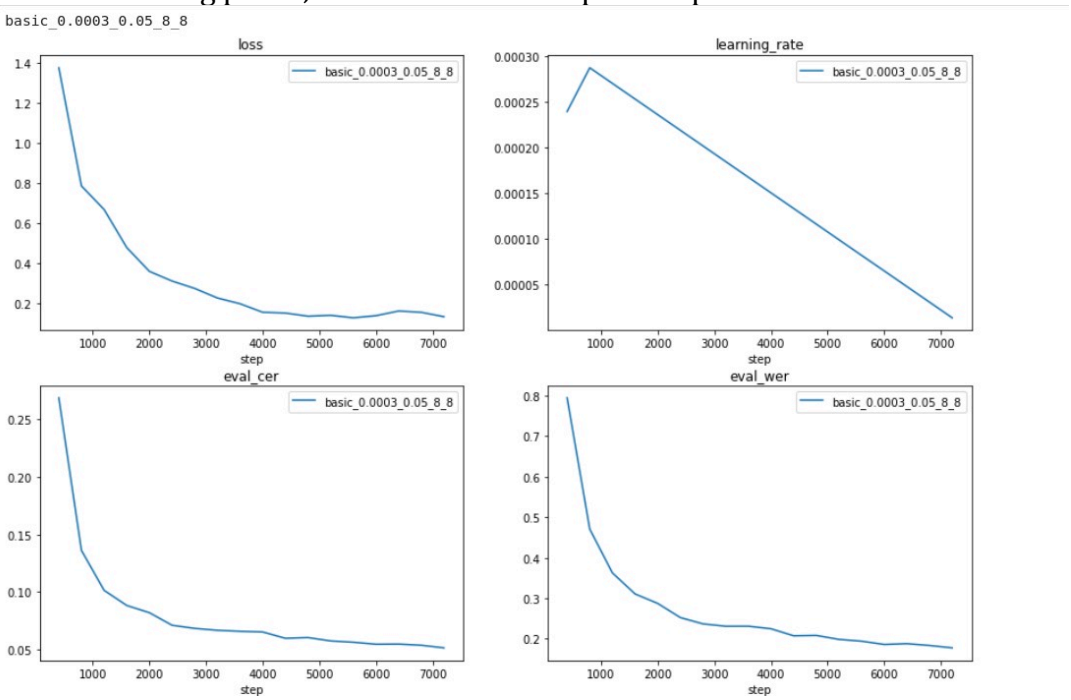
One of the biggest problems with models, is the overfitting. Which means that I trained a model on my own custom data, its accuracy rate is 90 (WER = 10), and I want to test the model on a new data. Sometimes, the model doesn't really learn the data, but it remembers it. Another problem is the environment. Sometimes a given data was received from a specific environment with its specific acoustic features, and when our model will get different data recorded from another environment, it might be failed to recognize the content properly results in a mistaken transcription.

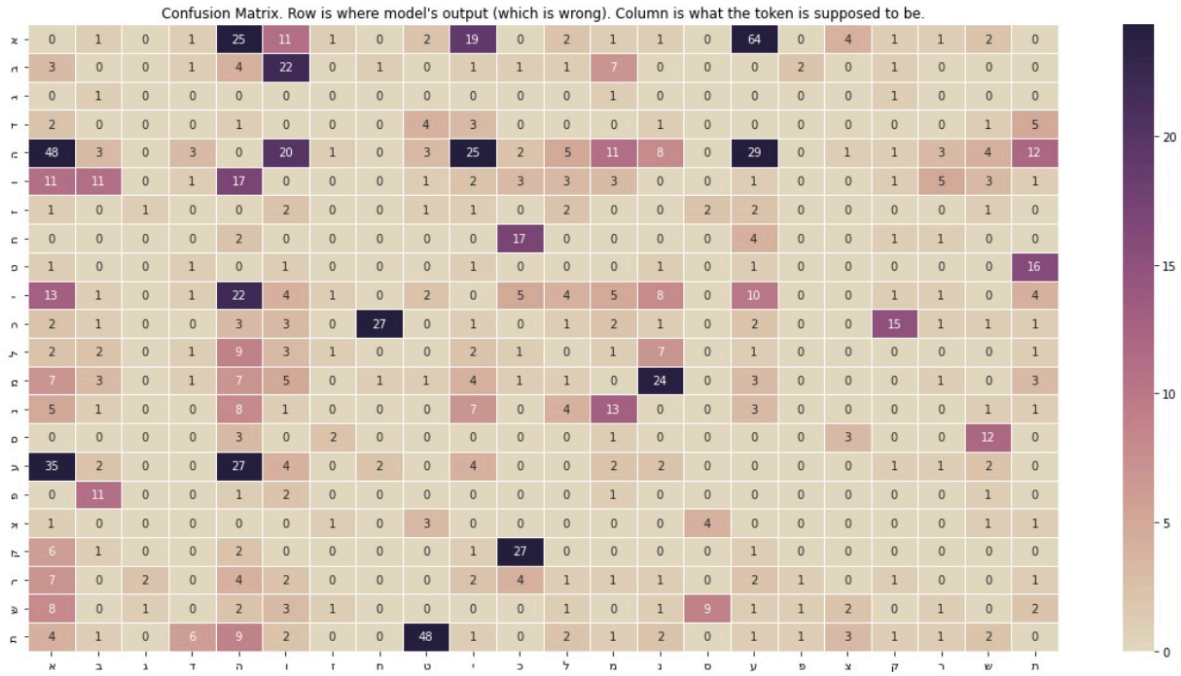
The best solution to this situation, is to hold a various data so by that the model can see various

URL: http://dx.doi.org/10.14738/tmlai.106.13515

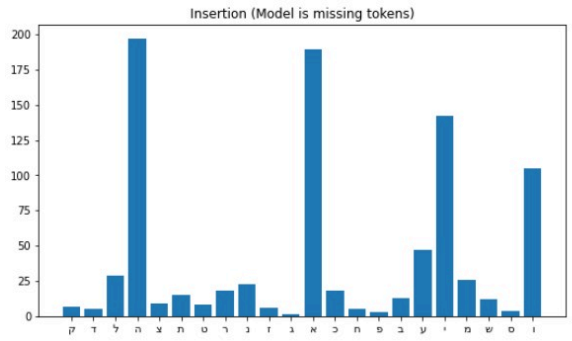
situations. Most of the times, dropout is also done in such kind of models (we used the mask instead). Dropout is a process of hiding a part of the wave. We can also hide some of the layers features or the layers themselves as well. The main goal of this operation is making sure that even if we have a non-various data, the model will see the data in some various ways, which makes the same effect of seeing a various data. In our training phase, we used the mask, as written above.

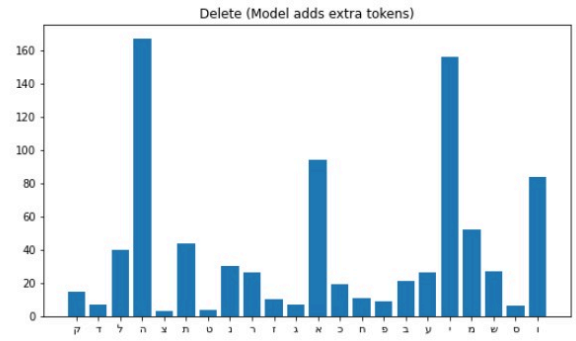
After the model training phase, we evaluated its important parameters:





After training the model, we also made a confusion matrix with the letters switches, insertions and omissions.





CONCLUSION

In this paper, we present the current state-of-the-art models for Automatic Speech Recognition due to a self-supervised training implemented on Hebrew language. The motivation behind using self-supervised learning is that even though we wouldn't probably get the accuracy rates as if we choose a supervised learning, we still can achieve amazing results with relatively low amount of data.

This way of training allows us to train a model on unlabeled data (or to use a pre-trained model, which is always more accessible. It's goal in the first unsupervised phase is to learn some good representations from raw audio samples, which can be useful for speech recognition tasks, without using any label data. Then, the model can be fine-tuned on a particular dataset for a specific purpose. It means that our involvement really occurs in the last layers of the model. This kind of training proved to be very powerful. We present complete framework from model to practice with simulations and training model and present an impressive result on Hebrew.

References

Xin Feng, Youni Jiang, Xuejiao Yang, Ming Du, and Xin Li. Computer vision algorithms and hardware implementations: A survey. Integration, 69:309–320, 2019. 1

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020. 1

Li Shen, Laurie R Margolies, Joseph H Rothstein, Eugene Fluder, Russell McBride, and Weiva Sieh. Deep learning to improve breast cancer detection on screening mammography. Scientific reports, 9(1):1–12, 2019. 1

M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. In Proc. of ACL, 2018.

V. Pratap, A. Hannun, Q. Xu, J. Cai, J. Kahn, G. Synnaeve, V. Liptchinsky, and R. Collobert. Wav2letter++: A fast open-source speech recognition system. In Proc. of ICASSP, 2019.

P. Bachman, R. D. Hjelm, and W. Buchwalter. Learning representations by maximizing mutual information across views. In Proc. of NeurIPS, 2019.

D. Jiang, X. Lei, W. Li, N. Luo, Y. Hu, W. Zou, and X. Li. Improving transformer-based speech recognition using unsupervised pre-training. arXiv, abs/1910.09932, 2019.

W. Wang, Q. Tang, and K. Livescu. Unsupervised pre-training of bidirectional speech encoders via masked reconstruction. arXiv, 2020.

URL: http://dx.doi.org/10.14738/tmlai.106.13515