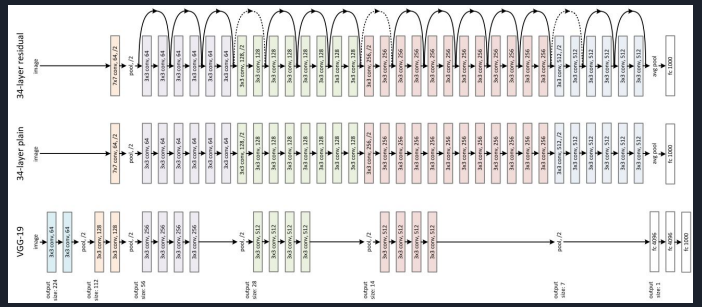
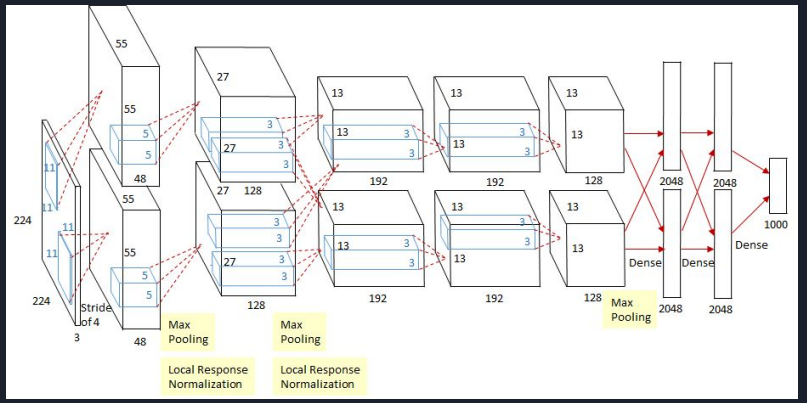
X-Vector embedding

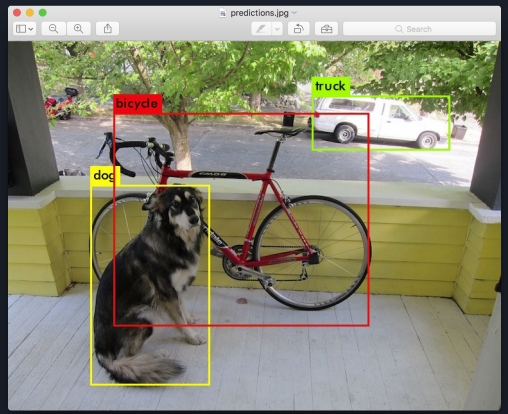
A modern approach to vectorize speaker identity.

The need for embedding

The state of the art of image recognition



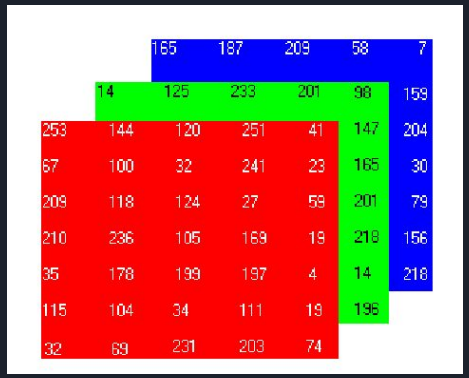




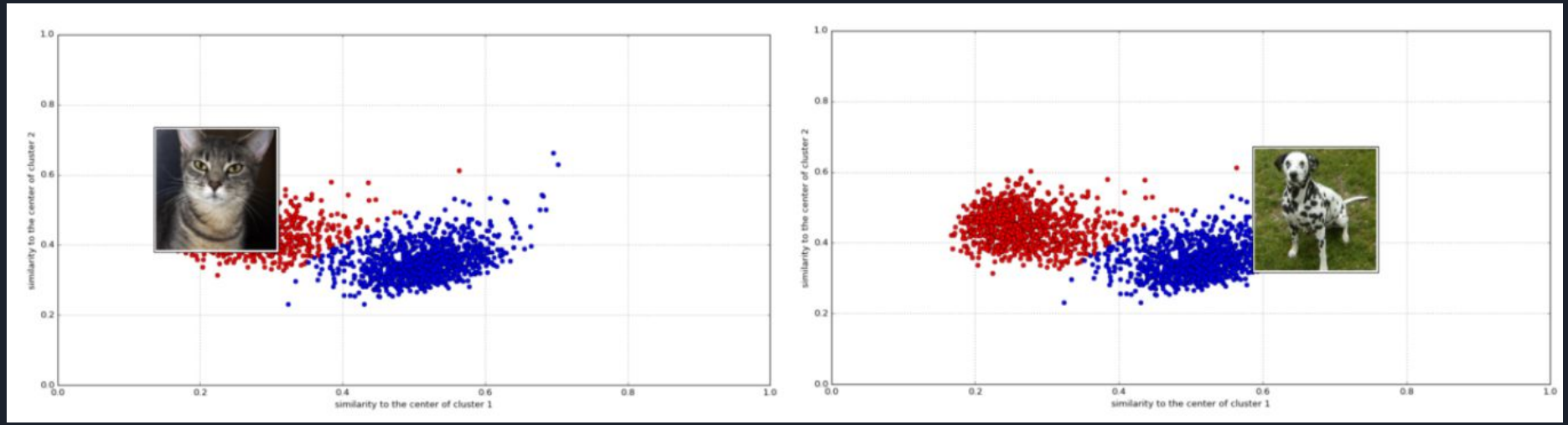
What are the input features ?

Well the image itself !

and we are done....



But what if we would like to cluster the images ?



One step further....

let's say now that's our input is a text

one hot encoding ?

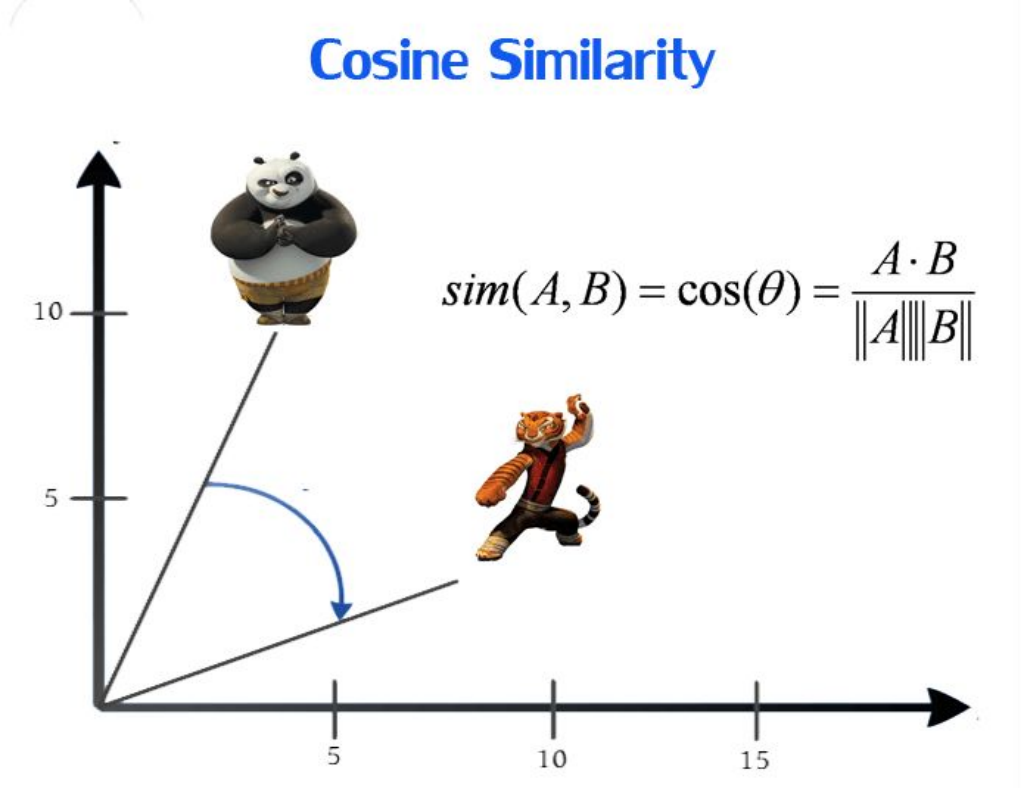
Color
Red
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1

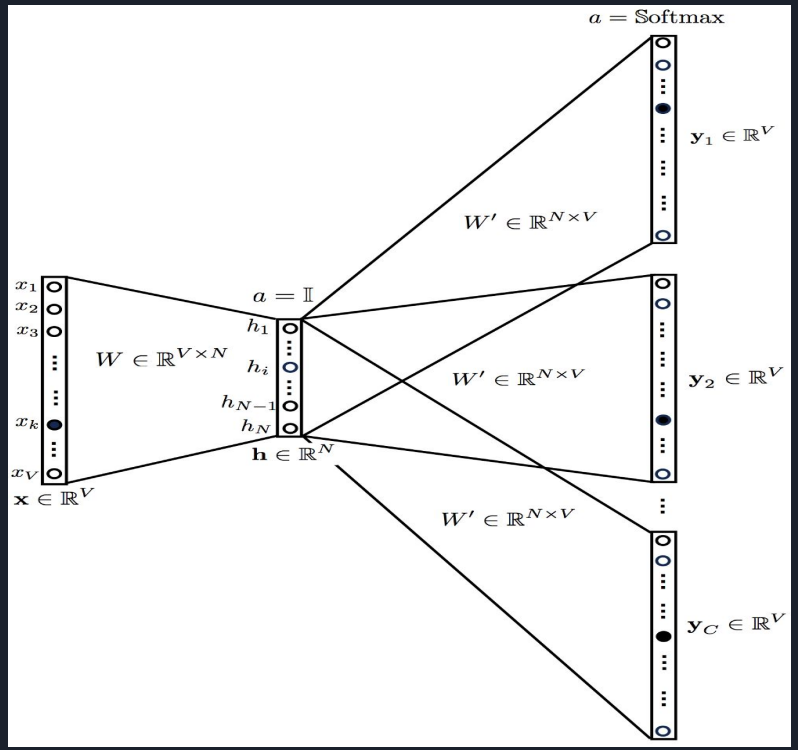
we just lost the semantic context of a word....

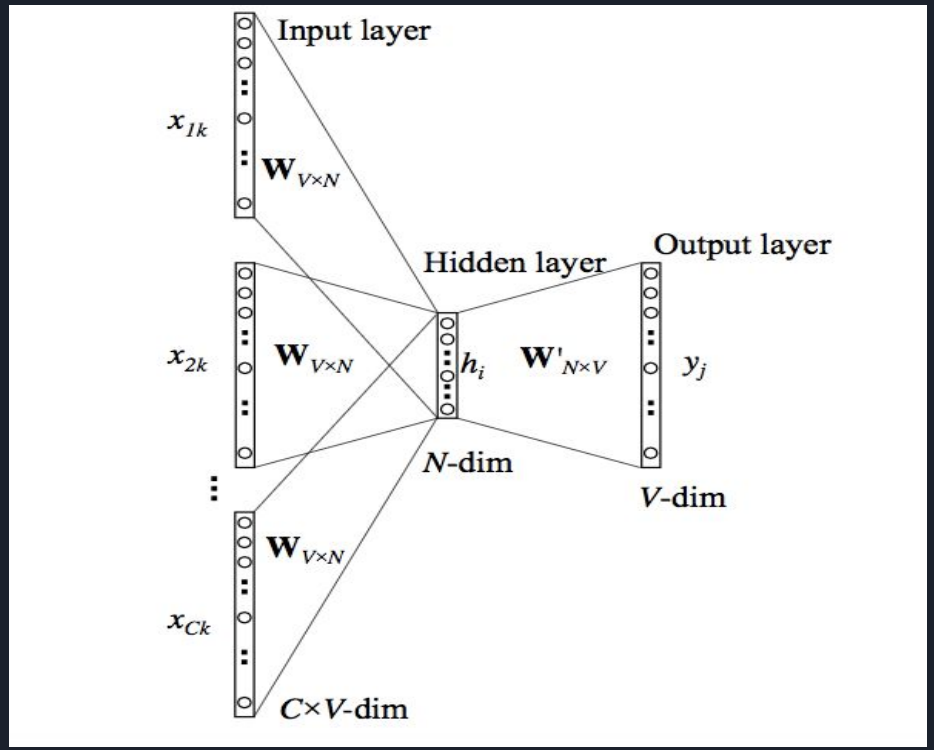
Our objective is to have words with similar context occupy close spatial positions.

Mathematically, the cosine of the angle between such vectors should be close to 1, i.e. angle close to 0.

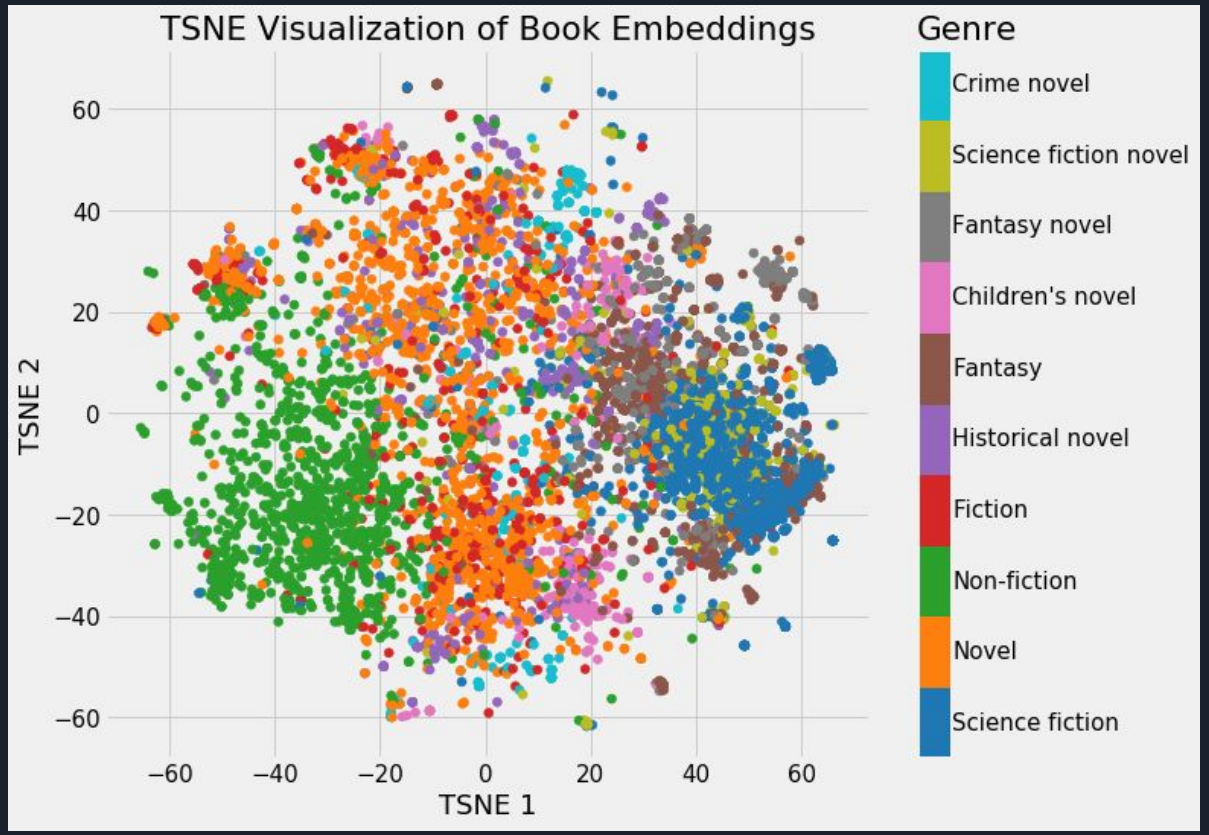


Word2vec came to help





The need for fixed size embedding

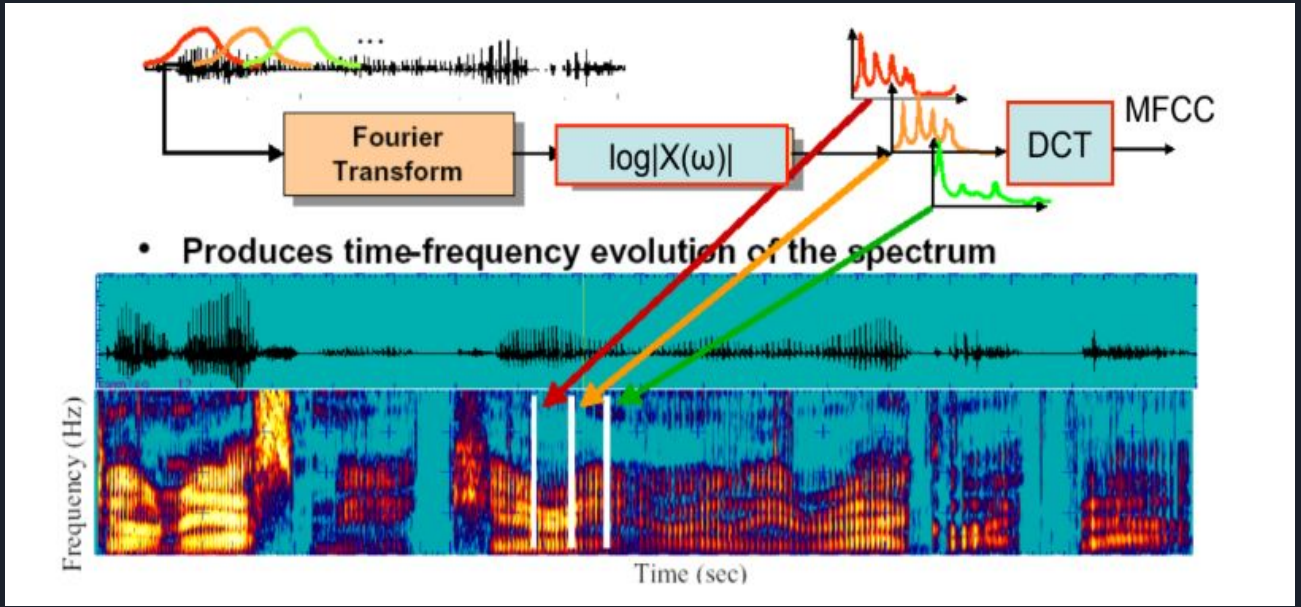


And now for speech embedding

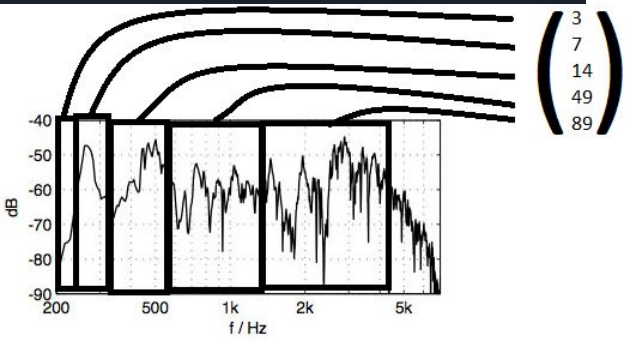
The first line of speech features

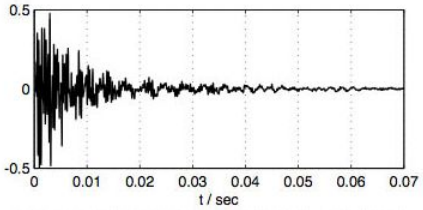
Speech is a time-varying signal conveying multiple layers of information

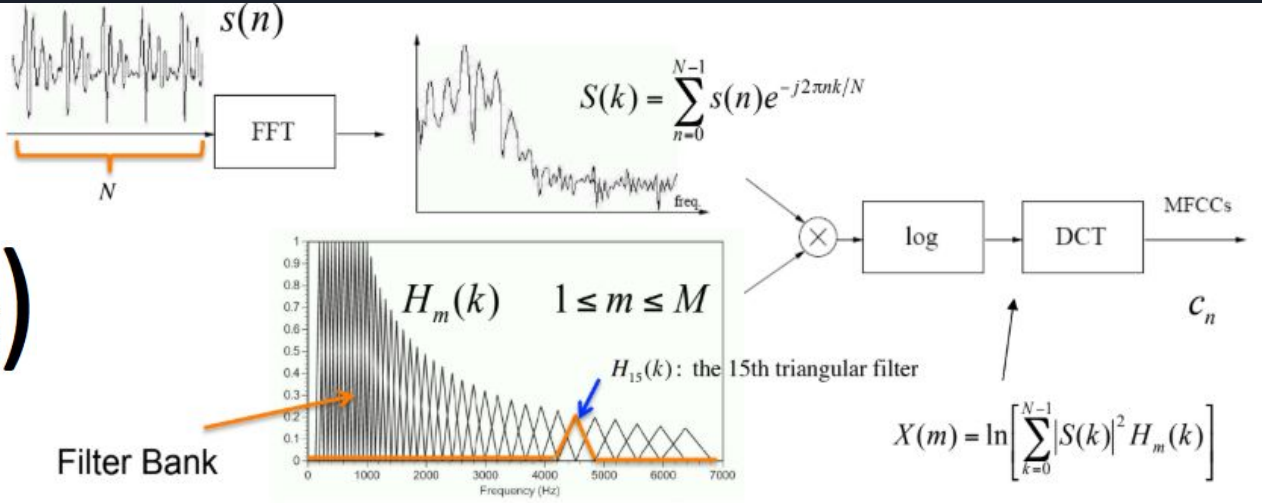
- Words
- Speaker
- Language
- Imotion



MFCC - Mel-frequency cepstrum







$$\text{MFCC}_n = c_n = \sum_{m=1}^{M} \cos\left[n\left(m-\frac{1}{2}\right)\frac{\pi}{M}\right] X(m) = \mathbf{d}_n^T \vec{X}, \qquad 0 \le n \le P$$

$$\Rightarrow \text{MFCC vector} = \begin{bmatrix} c_0 & c_1 & \cdots & c_P \end{bmatrix}^T = \mathbf{D}\vec{X}$$

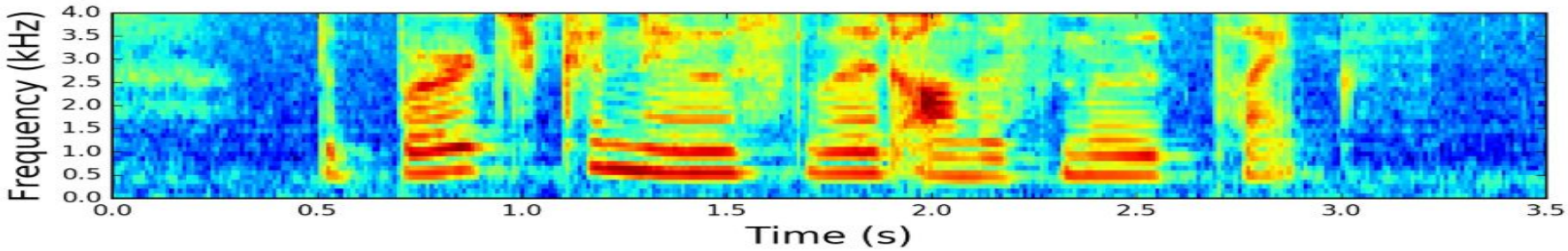
$\mathbf{D}$: DCT Transformation matrix $[P \times M]$

M : No. of triangular filters in the filter bank, typically $20 \sim 30$

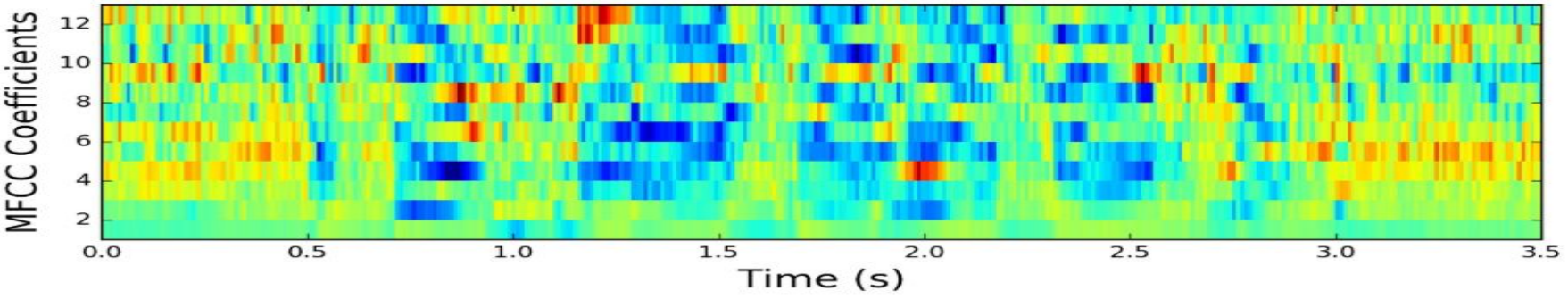
P : No. of cepstral coefficients, typically 12

c_0 : Logarithm of energy of the current frame

MFCC



Spectrogram of the Signal

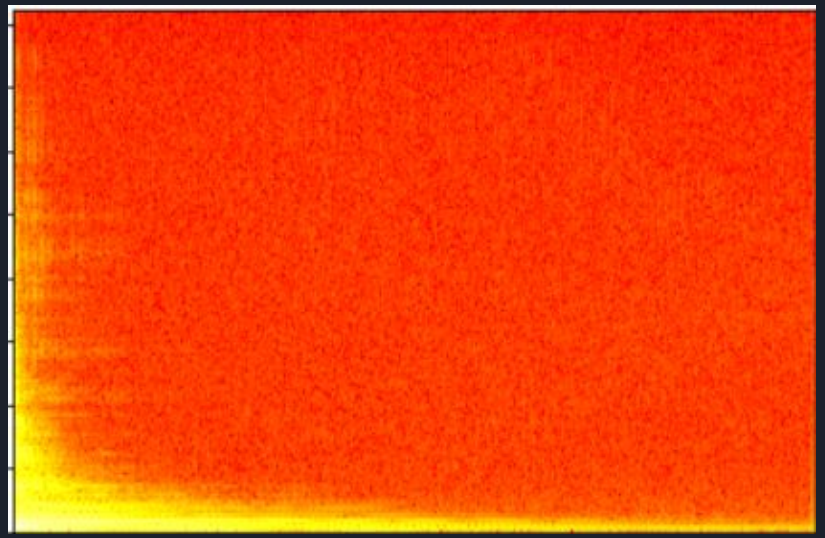


Normalized MFCCs

MFCC - Intuition

each musical instrument produces unique “fingerprint”

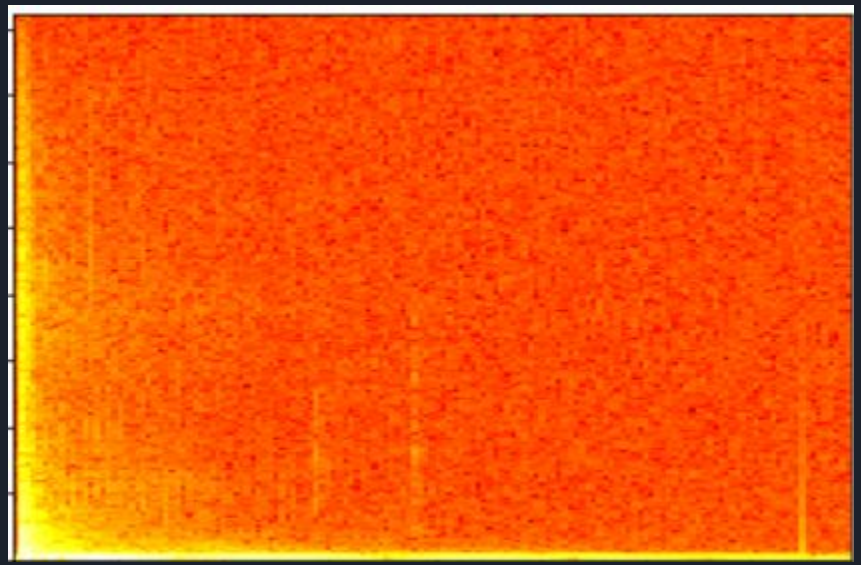
Acoustic guitar spectrogram vs MFCC



MFCC - Intuition

each musical instrument produces unique “fingerprint”

Bass drum spectrogram vs MFCC



for comparison

And now for the embedding part...

The pre Deep Learning era - The **I-vector**

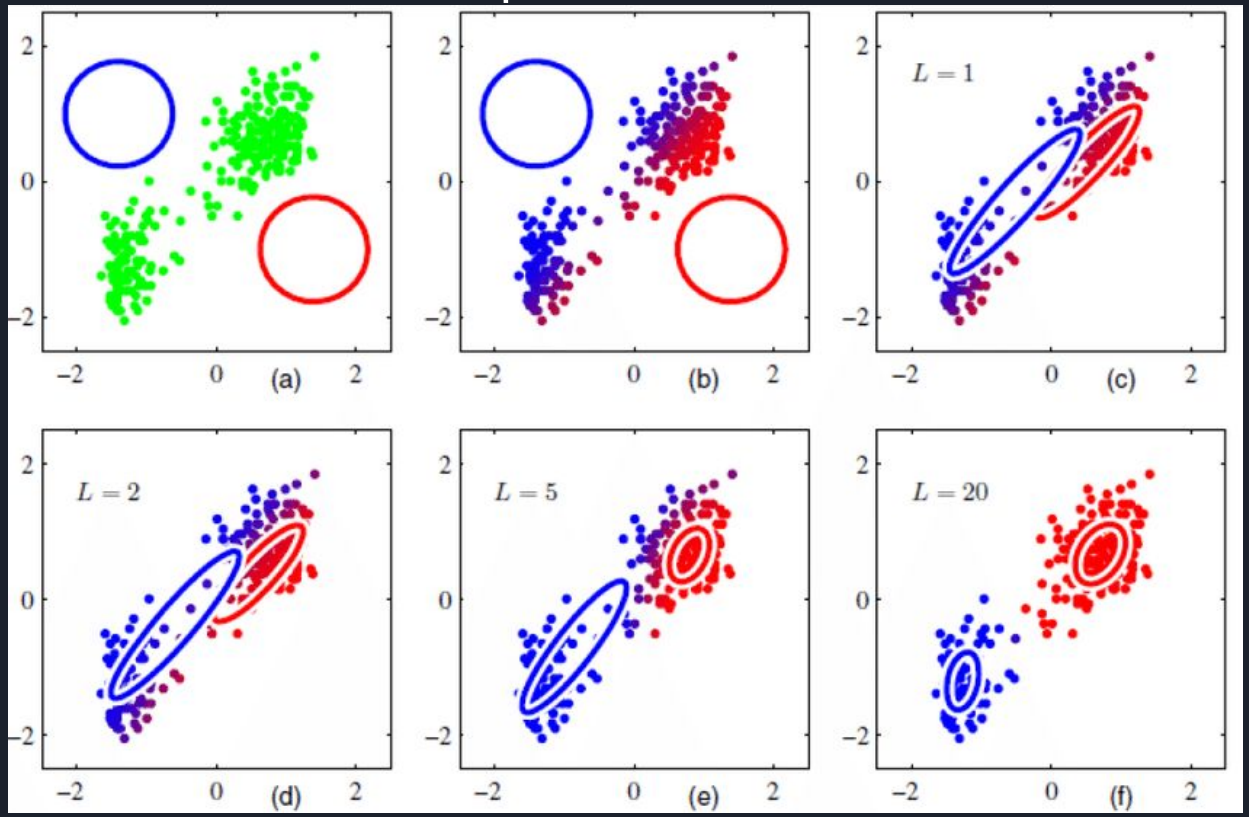
Unsupervised embedding model

each point is a mfcc

UBM - GMM – Universal Background Model

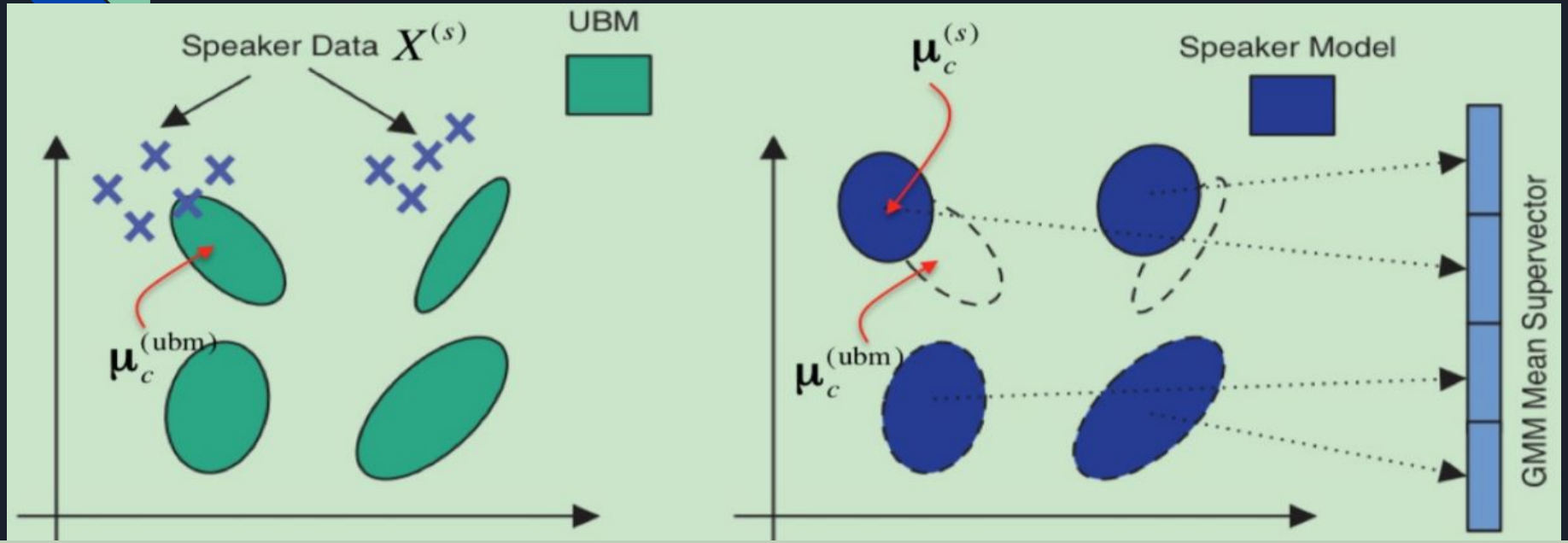
Use MFCC as features

Use 2048 (standard value) Gaussians to describe each of the features



Adaptive GMM - Extract

Adapt the UBM to the speaker using AGMM (using EM or MAP adaptation)



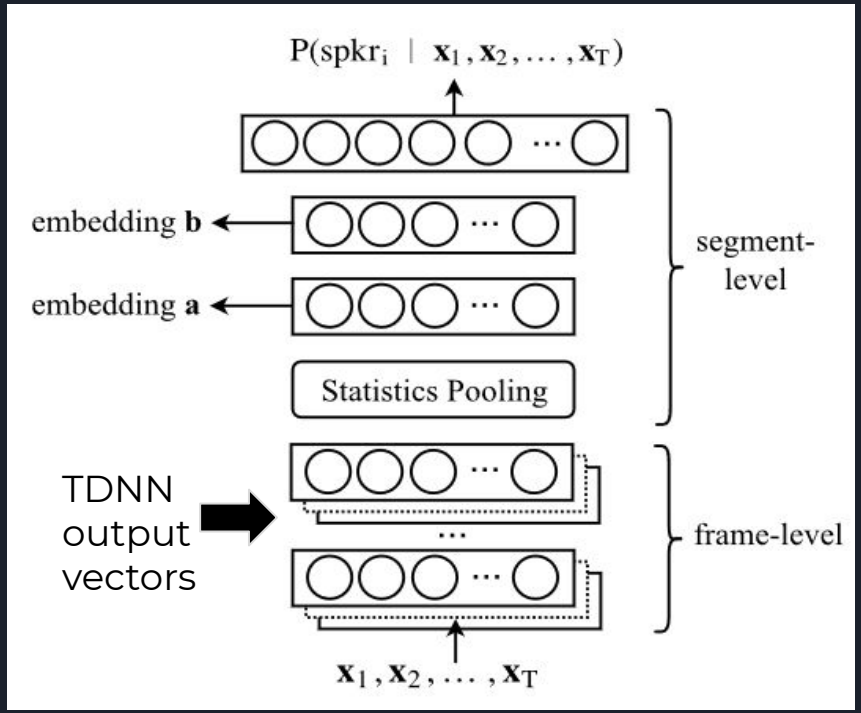
The 65k parameters vector aka Super Vector,
via dimensionality reduction the commonly **I-vector** is produced (using un/supervised methods) Ville Vestman, Tomi Kinnunen 2018

The X-vector

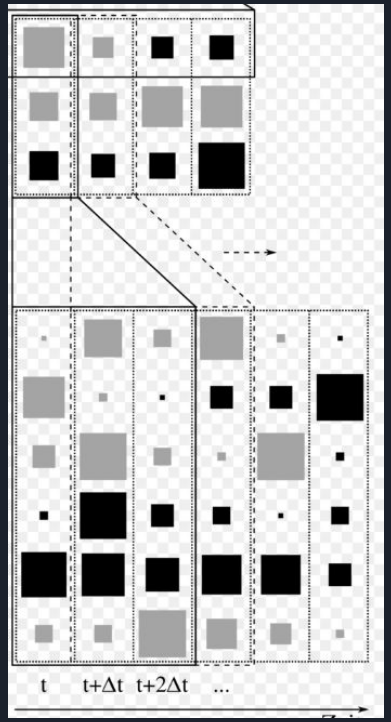
X - vector architecture

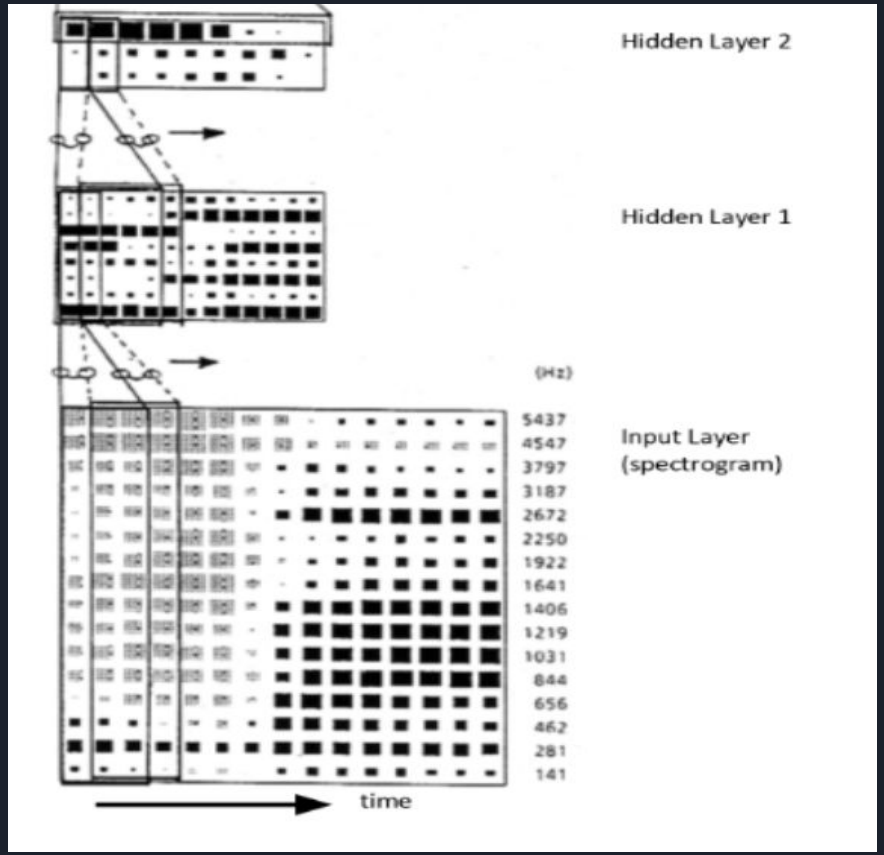
x-vector consists of three main parts:

- TDNN layers
- Stats pooling
- Fully connected layers

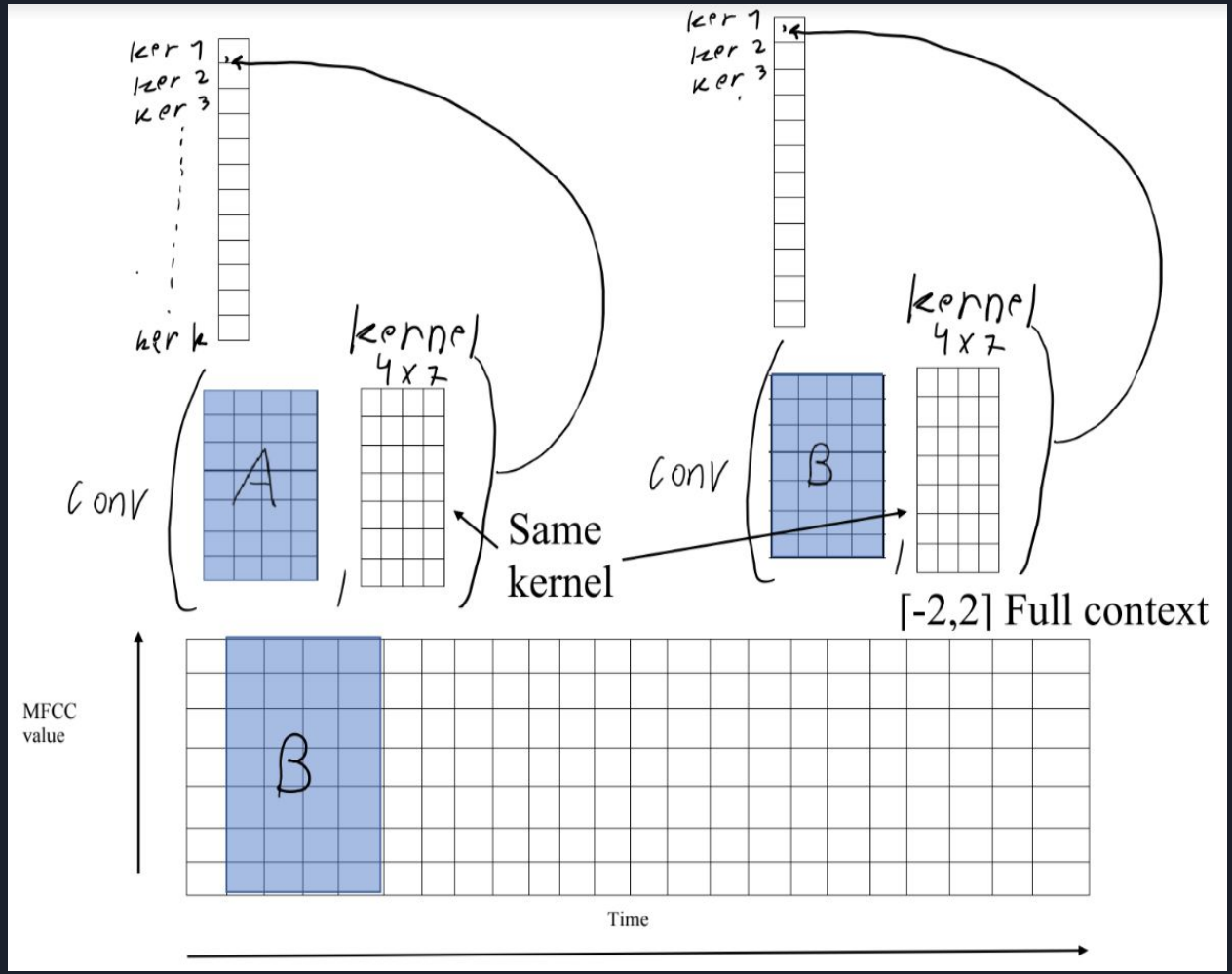


The TDNN - Time delay neural net

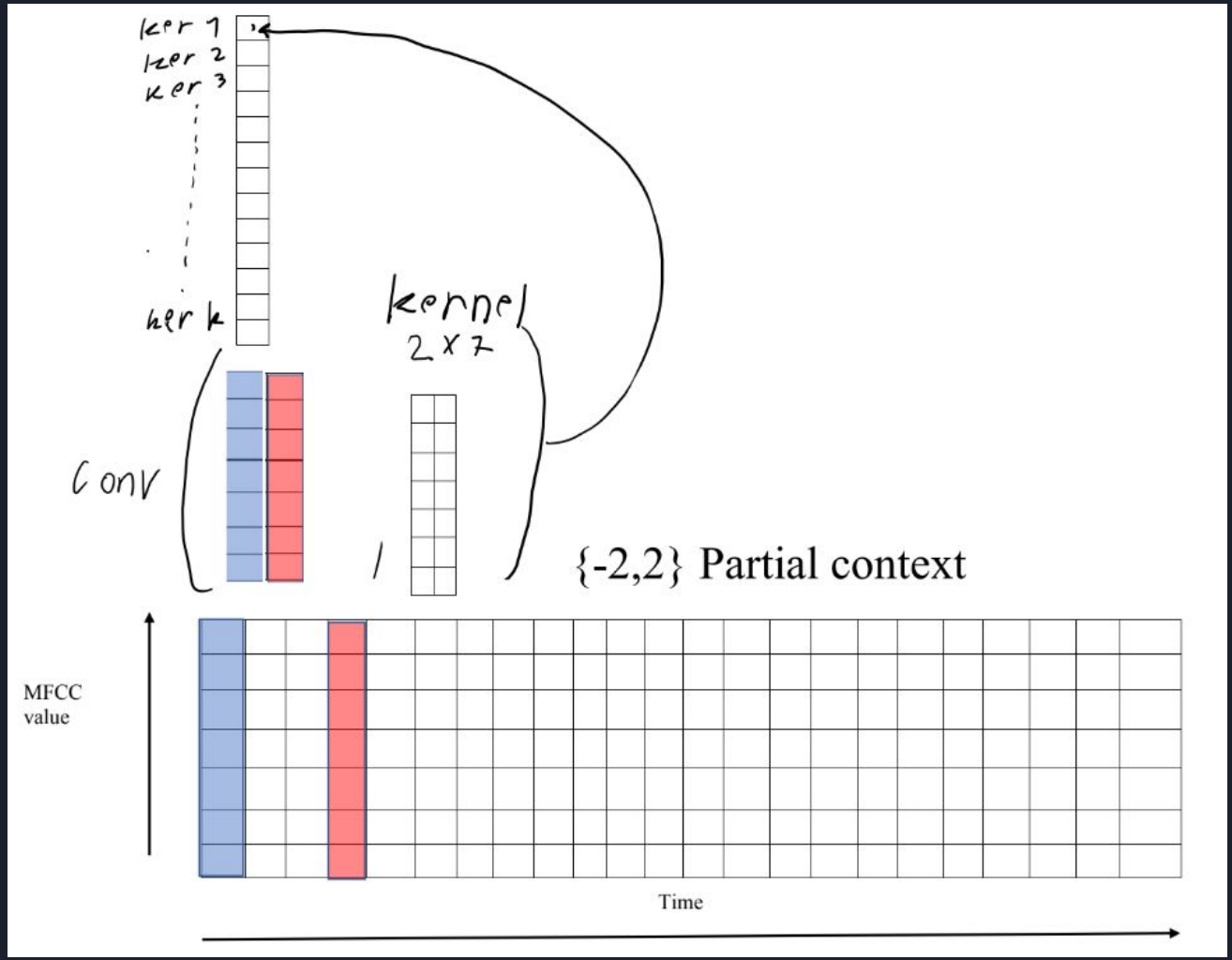




Implementation using 1d convolution layer



The TDNN introduced the partial context idea

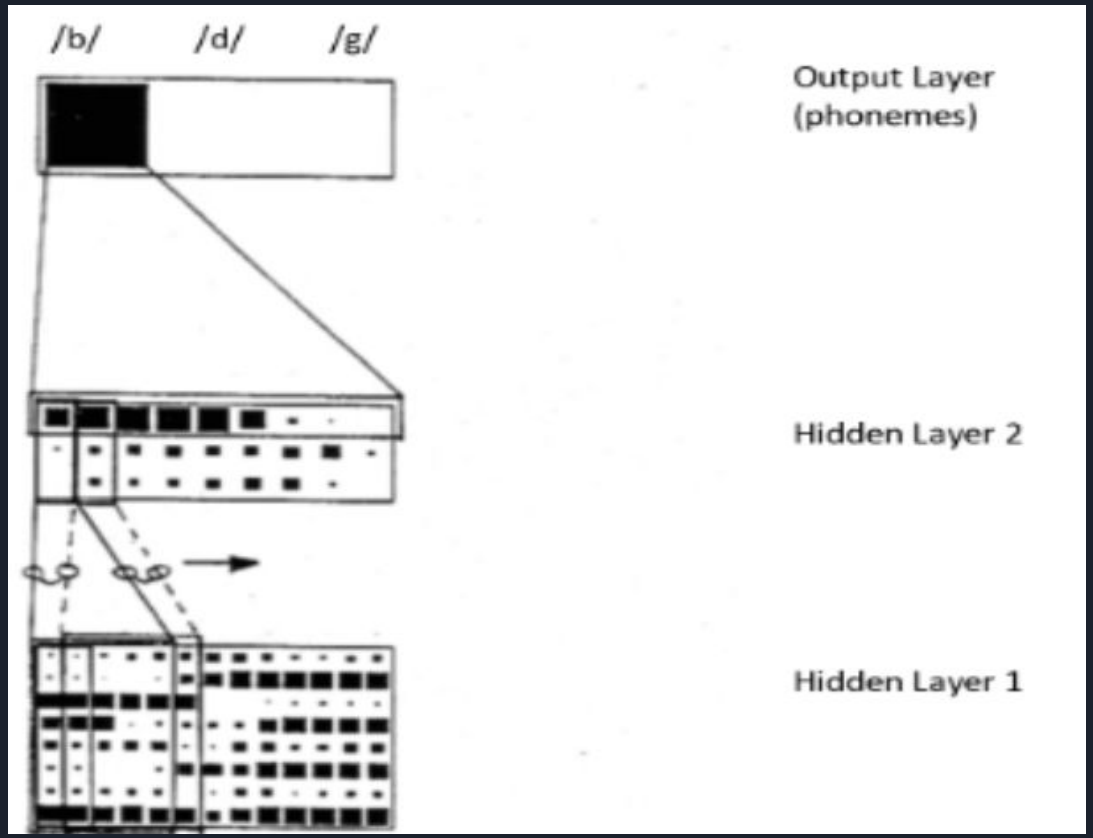


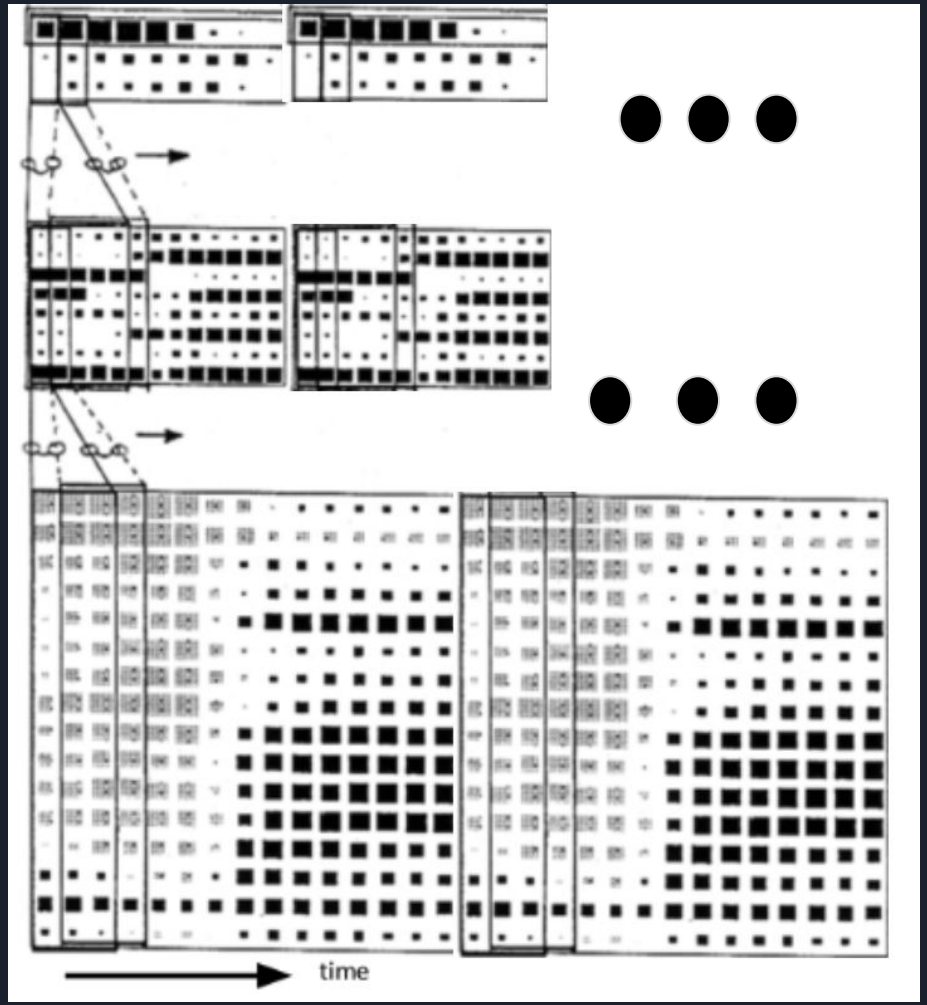
TDNN usage beside X-vectors

The TDNN could be used for phonemes recognition.
ALEXANDER WAIBEL et al

And even for keyword spotting
Ming Sun , David Snyder et al

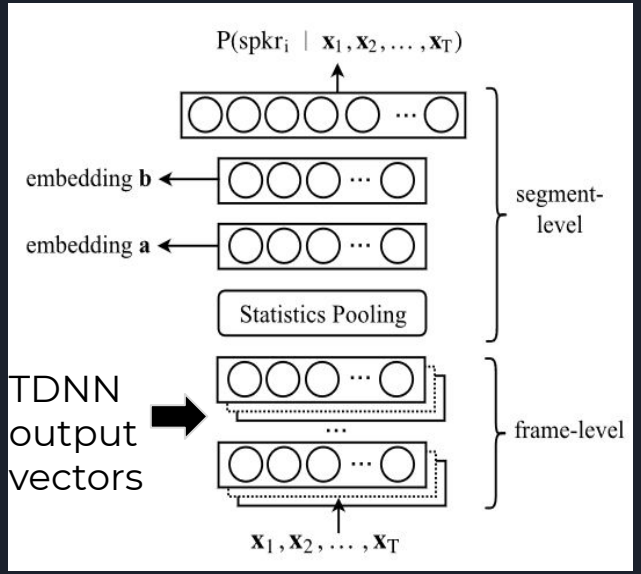
Depending on the output layer of our choice...



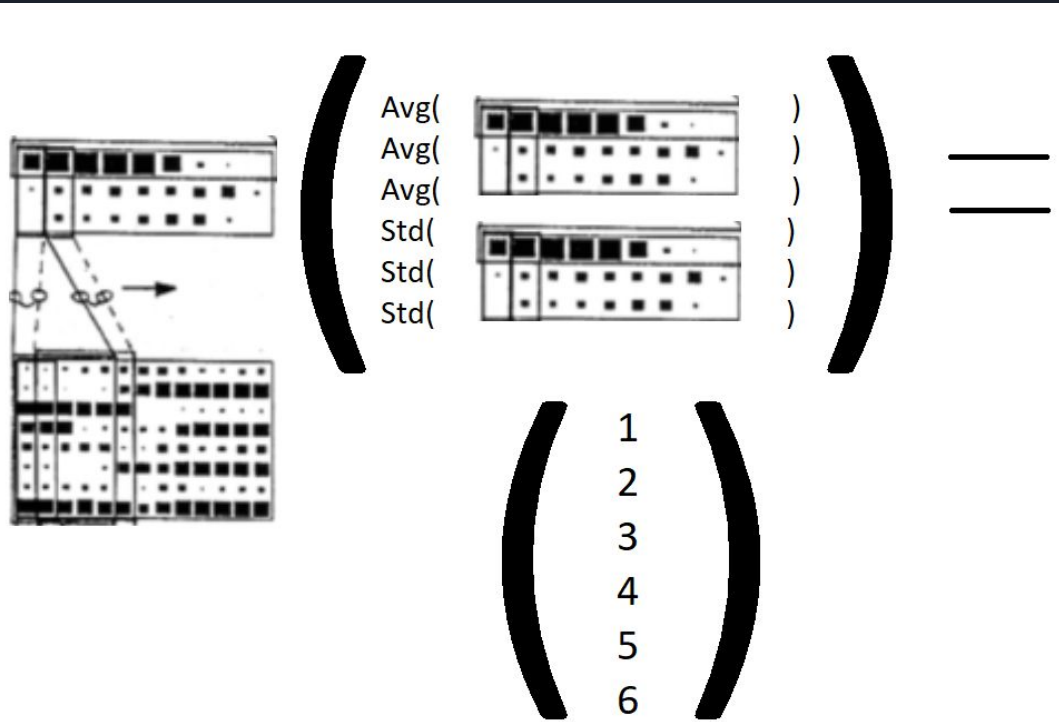


Depending on the input length we will receive **dynamic length** end layer.

Which means we still don't have the **fixed size** vector that we wanted



Statistics - pooling

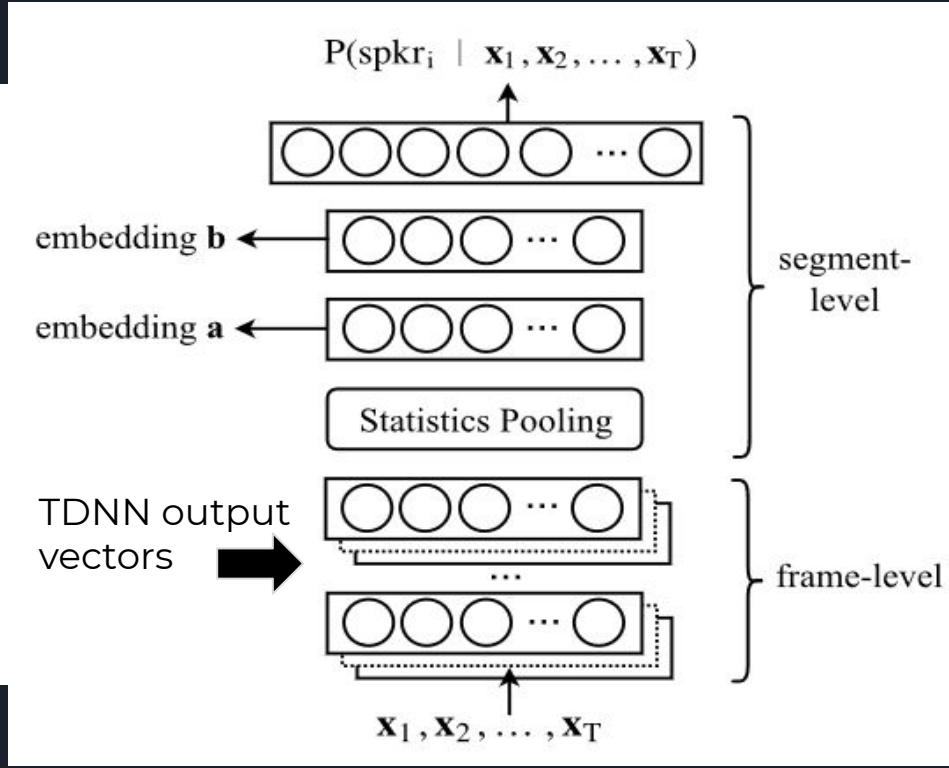


Simply take the average and the standard deviation for each channel of the final TDNN output and concatenate them

And now we got our desired **fixed length** vector

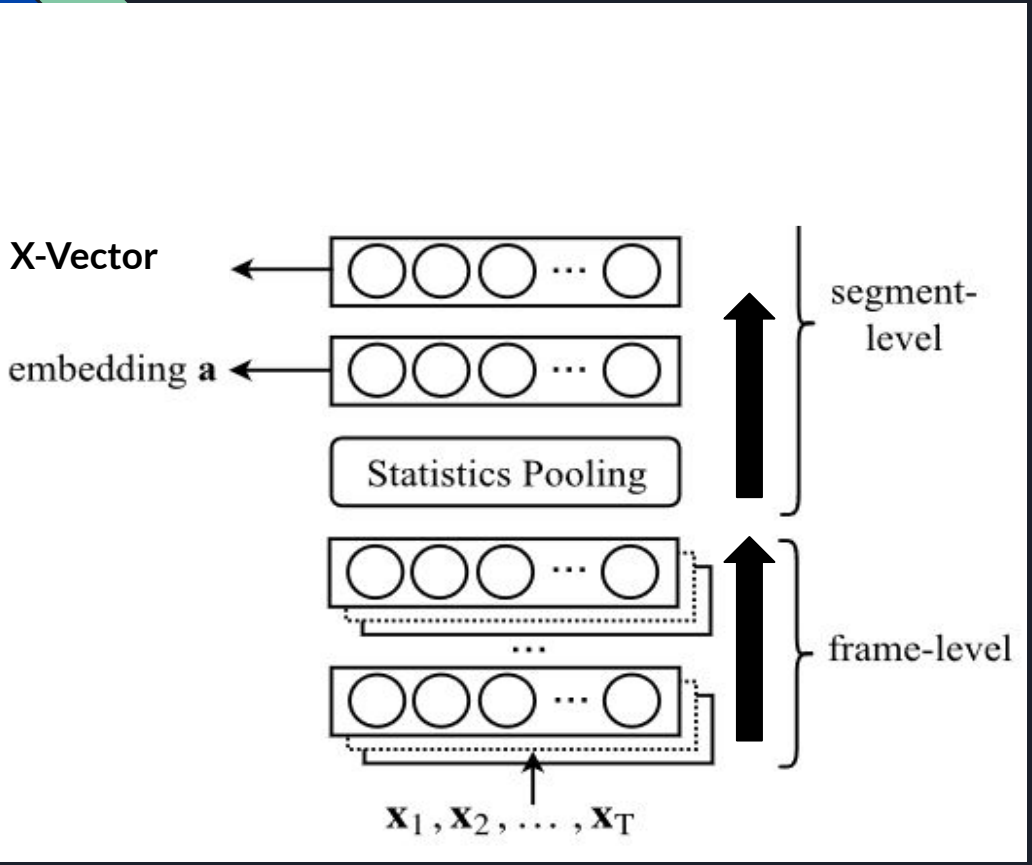
Two final steps and we have out **x - vector**

Layer	Layer context	Total context	Input x output
frame1	$[t-2, t+2]$	5	120x512
frame2	$\{t-2, t, t+2\}$	9	1536x512
frame3	$\{t-3, t, t+3\}$	15	1536x512
frame4	$\{t\}$	15	512x512
frame5	$\{t\}$	15	512x1500
stats pooling	$[0, T)$	T	1500Tx3000
segment6	$\{0\}$	T	3000x512
segment7	$\{0\}$	T	512x512
softmax	$\{0\}$	T	512xN



The stats pooling layer output is than sent to two Fully - connected layers

Chopping the network head



After learning the network parameters we chop off the softmax layer, and forward propagate an input signal

Demonstration

The most common public implementation of X-vectors is using the Kaldi framework

Until now...

Pytorch implementation of X-vectors

```
context = [-2,2]
input_dim = 24
output_dim = 512
net1 = TDNN(context, input_dim, output_dim, full_context=True)

context = [-2,1,2]
input_dim = 512
output_dim = 512
net2 = TDNN(context, input_dim, output_dim, full_context=False)

context = [-3,1,3]
input_dim = 512
output_dim = 512
net3 = TDNN(context, input_dim, output_dim, fu

context = [1]
input_dim = 512
output_dim = 512
net4 = TDNN(context, input_dim, output_dim, fu

context = [1]
input_dim = 512
output_dim = 1500
net5 = TDNN(context, input_dim, output_dim, fu
```

Layer	Layer context	Total context	Input x output
frame1	$[t-2, t+2]$	5	120x512
frame2	$\{t-2, t, t+2\}$	9	1536x512
frame3	$\{t-3, t, t+3\}$	15	1536x512
frame4	$\{t\}$	15	512x512
frame5	$\{t\}$	15	512x1500
stats pooling	$[0, T)$	T	1500Tx3000
segment6	$\{0\}$	T	3000x512
segment7	$\{0\}$	T	512x512
softmax	$\{0\}$	T	512xN

So how exactly is TDNN implemented ?

```
# Perform the convolution with relevant input frames
for c, i in enumerate(valid_steps):
    features = torch.index_select(x, 2, context+i)
    # print ('features taken:{}'.format(features))
    xs[:,:,c] = F.conv1d(features, kernel, bias=bias)[:,:,0]
return xs
```

How the stats pooling and the final layers looks like ?

```python
class StatsPooling(nn.Module):
    def __init__(self):
        super(StatsPooling,self).__init__()

    def forward(self,varient_length_tensor):
        mean = varient_length_tensor.mean(dim=1)
        std = varient_length_tensor.std(dim=1)
        return torch.cat((mean,std),dim=1)
```

```python
class FullyConnected(nn.Module):
    def __init__(self):
        super(FullyConnected, self).__init__()
        self.hidden1 = nn.Linear(3000,512).double()
        self.hidden2 = nn.Linear(512,512).double()

    def forward(self, x):
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        return x
```

The Full network

```
26 SP = StatsPooling()
27 FC = FullyConnected()
28 Final = nn.Linear(512,4).double()
29 net = nn.Sequential(net1,net2,net3,net4,net5,SP,FC,Final)
```

```
1 clases = ['Trump','Macron','Gaga','Clinton']
```

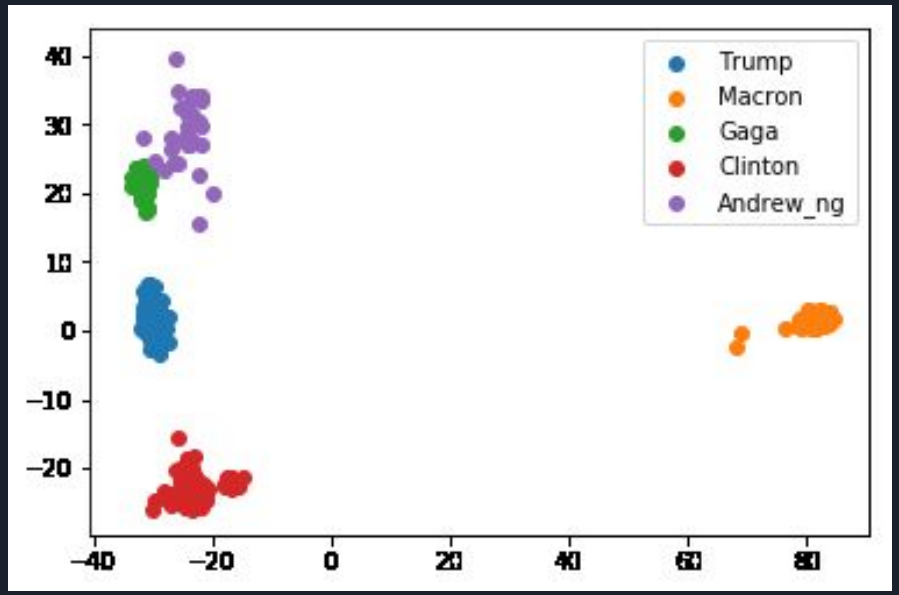
```
mel size: 24
sequence length: 400
output shape: torch.Size([5, 396, 512])
mel size: 512
sequence length: 396
output shape: torch.Size([5, 392, 512])
mel size: 512
sequence length: 392
output shape: torch.Size([5, 386, 512])
mel size: 512
sequence length: 386
output shape: torch.Size([5, 385, 512])
mel size: 512
sequence length: 385
output shape: torch.Size([5, 384, 1500])
```

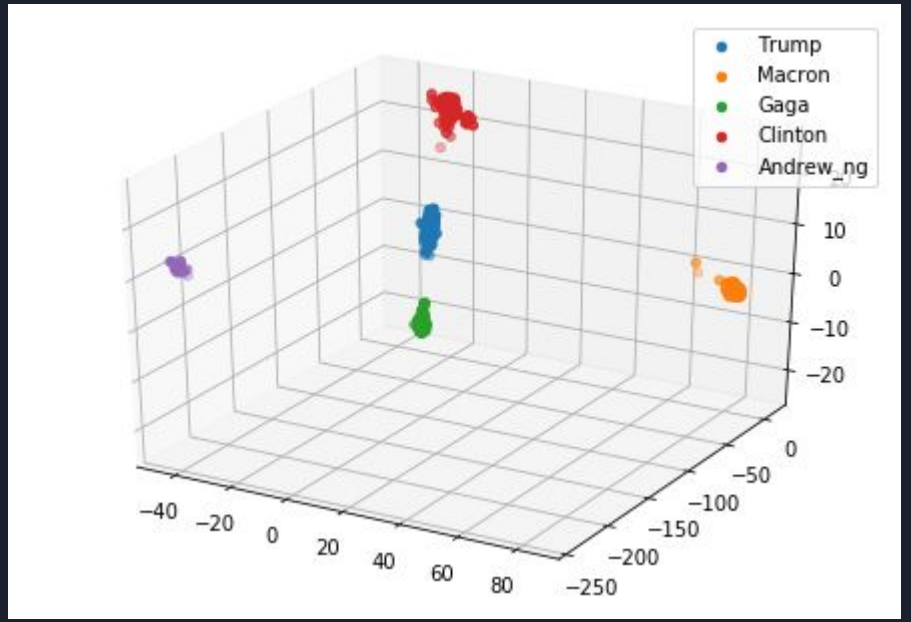
```
loss: 0.02339517055584004 step took 22.59490942955017
output:
tensor([[  6.6202,  -6.6046,   0.9639,  -0.9110],
        [  0.8131,  -3.1778,  -2.1677,   4.8661],
        [  5.7950,  -4.4012,  -1.2745,  -0.0496],
        [  4.0205, -12.8199,   8.0885,   1.0369],
        [  1.5110,  -2.8568,  -2.4932,   4.1227]], device='cuda:0',
       dtype=torch.float64, grad_fn=<AddmmBackward>)
 labels:
tensor([[1, 0, 0, 0],
        [0, 0, 0, 1],
        [1, 0, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]], device='cuda:0', dtype=torch.uint8)
```

The **x-vector** embedding

```
net_xvec = nn.Sequential(net1,net2,net3,net4,net5,SP,FC)
net_xvec.cuda()
```

LDA dimensionality reduction for visualization





https://github.com/Dannynis/xvector_pytorch

Questions ?