# Deep learning approach for keystroke recognition

Stanciu Andrei Călin

**Abstract**

This is a technical report in which variations of the same deep neural network aer used for keystroke recognition between two persons, in a free-text scenario, by using only key press timestamps. The accuracy obtained is around 80%, with high variance around this result, which suggests the model is clearly able to learn patterns from the keystroke sequences, although the model's capacity to be generalised remains unclear due to the small-scale context in which it was trained and evaluated. The implementation for both the data collector and the model, and the data can be found at [15].

## 1 Introduction and related work

Although maybe not as popular or used in practice as other biometrics-based authentications, keystroke biometrics were thoroughtly investigated over the course of the last few decades. This method of authentication can be viewed, from the scope of the data being used, as being based on *fixed-text* scenarios, or *free-text* scenarios. Another distinction that can be made is between desktop keystrokes on a "physical" keyboard and mobile "touchscreen" keystrokes - a detailed analysis on most of the existing literature can be found here [1]. From this point of view, the main resources on which this report is based on is [2] and [3]. In [2], the authors used a LSTM [5] based model that was trained with softmax loss, and then in comparison with contrastive and triplet loss, which generally yielded better results [2, p. 9]. In [3], the emphasis is placed on using a net which consists of using convolution layers to create a more meaningful representation of the keystroke sequence, and then pass it to GRU [4] layers to process them further.

The main focus of this experiment is to test whether a (relatively simple) deep neural net can be used to discriminate between different people, based on limited keystroke metrics. More precisely, only the timestamps of key presses are taken into account (the accuracy of time measurements and other considerations will be detailed in further sections). Although the implementation had been built with generalization and extensive parametrization in mind, the actual tested models are more limited, and the classification itself is only binary.

The rest of this article is organised as follows: the dataset that is used and all its properties are detailed in section 2, the (final) model and some of its variations are presented in section 3, and the results of training and validation are presented in section 4. Before going further, it is also worth mentioning that the source code and the entire dataset used are open source, and thus the results that will follow can be replicated in the same context as those presented here.

## 2 Dataset

### 2.1 Data collection process

The dataset was collected on linux machines, by interacting directly with the kernel data structures. This provided very accurate timestamp series: each timestamp consists of seconds since epoch, and the microsecond offset in that second. All of those are colected inside binary files, which are further processed when loading the data. A timestamp represents the moment a key is pressed. Other metrics were taken into account, such as the hold duration of the key or the last-release to next-press delay, which are taken into account by most of the research in this area, including the previous sources [2, 3]. Ultimately, only the key press moments were used, and, if the presumption that *when multiple metrics are fed into the model, the model is able to ignore those who do not correlate with the class from which that input comes from* holds, the "key press - only" results can be seen as a lower bound for the "multiple keystorke metrics" results - this remains to be validated through additional experiments, though. One other information that could have been extracted is the value of the key being pressed. This was explicitly ruled out, because of additional privacy concerns (although detecting who is typing can be considered a privacy threat by itself, disclosing sensitive information would make the dataset subject to various privacy regulations and restrictions); besides that, the minimalist argument applied to other metrics can be made here. Still, other researchers proceeded in successfully collecting the keys presed, without posing a privacy threat to the subjects whose keystrokes were being recorder - one such example, that also focused on free-text scenarios is the Clarkson II dataset [6].

Regarding the context in which those keystroke timestamps were collected, both classes that were used have timestamps collected from typing mostly python code, and the rest being text in natural language or other programming languages and any other activity that uses a physical keyboard.

One might remark that those circumstances do not fully represent a "free-text" scenario. Still, those differences might themselves help the model differentiate between classes, scenario which corresponds to real-world circumstances. This also leaves room for more experiments, that could shift their attention entirely on differentiating between natural typed text and programming languages typed text, but this is beyond the scope of the current experiment.

## 2.2 Data preprocessing

There are a few steps executed to process the raw timestamps collected (seconds, microseconds) into the data that is fed to the model.

The first step is the conversion of (seconds, microseconds) tuples into a single number, with the corresponding chosen precision: 0.01 milisecond precision, 0.1 milisecond precision, 1 milisecond precision and 10 miliseconds precision were all tried, with similar results. The final model uses a conservative 0.1 milisecond precision.

The second step is splitting the data in separate timeslices of fixed length, and the converting the absolute timestamps in relative timestamps to the beginning of the timeslice. The timeslice length was chosen empirically to be 30, although more values were tested (section 4 contains statistics for both timeslice lengths of 30 and 16). A new slice was created when the timeslice length surpassed the chosen value, or when the distance between two consecutive timestamps was bigger than some chosen value. This was implemented to eliminate "outlier" timeslices, that are semantically part of two or more different typing "sessions", rather than one continous typing streak, such as when writing a sentence or a programming language statement - the chosen value was 4 seconds. In addition, the stride used when splitting the continous sequence into timeslice-length sequence can be chosen to either be equal to the timeslice length itself, or half of it (for train data only). More formally, all of this can be expressed as:

- the raw timestamps, before partitioning:

$$T_0, T_1, T_2, ...T_k$$

- sequences with $l = len(timeslice)$ stride:

$$(T_1 - T_0, T_2 - T_1, ...T_l - T_{l-1}), (T_{l+1} - T_l, T_{l+2} - T_{l+1}, ...T_{2l} - T_{2l-1}), ...$$

- sequences with $l = len(timeslice)/2$ stride:

$$(T_1 - T_0, T_2 - T_1, ...T_l - T_{l-1}), (T_{l/2+1} - T_{l/2}, T_{l/2+2} - T_{l/2+1}, ...T_{3l/2} - T_{3l/2-1}), ...$$

The main goal was to increase the train data in a situation in which classic data augmentation techniques would not work. For example, when dealing with certain computer vision tasks, one could augment the data by rotating the images, increase / decrease contrast and so on. But in this situation, the human agent is not capable of detecting patterns in timeslices to be able to create changes in the data such that the corresponding class would be invariant to them. This augmentation would benefit the model in situations in which there exist correlations between two $t_i$ and $t_j$ such that $j > i + len(timeslice)/2$. In practice, a slight increase in the final accuracy can be observed (such comparison is presented in section 4), which would confirm this hypothesis.

The third step is to pertition the data according to the validation ratio, and also to have an equal number of overlapping / non-overlapping timeslices per class. This entire process can be summarised as follows:

$o_t^i$ - overlapping timeslices for class i (i.e. they overlap with the slices that would be obtained by splitting with stride equal to the length of the slice itself, beginning from $T_0$)

$o_f^i$ - non-overlapping timeslices for class i

$train_i$ - train data for class i

$validation_i$ - validation data for class i

$v_{ratio} \in (0, 1)$ - ratio validation:train

$$v_{count} = \lfloor v_{ratio} * min_i(|o_f^i|) \rfloor$$

$$train_i = o_t^i[: min_i(|o_f^i|)] + o_f^i[v_{count} : min_i(|o_f^i|)]$$

$$validation_i = o_f^i[: v_{count}]$$

where [:] and + denote list slicing and list concatenation.

## 2.3 Random data

It is worth mentioning that the implementation offers the posibility to include the *random class* to compete with human data. It was initially used for a binary classification "human vs random" to determine a baseline for the following "humans vs humans" classification. With a relatively basic convolutional architecture the validation accuracy reached 99%+ accuracy. This report will not include any more information regarding this classification, although (as with the other results) they can be replicated by running the source code on given data (or any other human data), vs automatically generated random data.

# 3 Model

One final model was found to perform the best out of many variations tried. The following subsection will present its structure, hyperparameters, and training methods, and then a few variations that also yielded satisfactory results will be presented. For each of those models, the next section will present statistics and discuss the results.

## 3.1 Base model

The model of choice is composed of 2 main parts, according to the supervised contrasting learning paradigm presented in [7]:

### 3.1.1 Encoder

The first component is the *encoder*, that is first trained at the beginning to create embedding vectors for samples (timeslices) from different classes. The loss, called *SupCon loss* [7, eq.2], is defined in the current implementation as:

$$\mathcal{L}_{supcon} = \frac{\sum\limits_{i \in I} \mathcal{L}_{supcon,i}}{|I|} = \frac{\sum\limits_{i \in I} \frac{-1}{|P(i)|} \sum\limits_{p \in P(i)} log \frac{exp(z_i * z_p / \tau)}{\sum\limits_{a \in A(i)} exp(z_i * z_a / \tau)}}{|I|}$$

Where $I$ is the set of indexes of the current batch, $A(i) \equiv I \setminus \{i\}$, $P(i) \equiv \{p \in A(i) | class(p) = class(i)\}$, $z_i$ is the feature vector of timeslice $i$ and $\tau$ is the temperature constant. There only difference between the above presented formula and the original one is that insead of summing over all losses from each element from the batch, the average is taken. This is also the approach from the demo presented here [13]. In essence, the encoder outputs the embedding vectors for each timeslice in the current batch, the cosine distance divided by temperature $\tau$ is calculated between each feature vector, and then softmax and crossentropy loss is applied, averaging at the end.

The encoder uses two main types of blocks:

- the LSTM cell with forget gate, which is already implemented in Keras framework ([14]), and is described by [10]
- a custom block that was inspired by both the residual blocks from ResNet [8] and the inception blocks from GoogleNet [9] - for simplicity, for the rest of this article, it will be called *inc1d*. Its structure is shown in figure 1. The rationale behind this custom block was to start from the residual block as described in [8, pg. 6] which was proven to be effective in computer vision tasks, and was hypothesised (in the same paper) to work even on non-vision tasks. In this experiment, the overall structure is taken, the 2D convolutions are replaced with 1D convolutions, and the shortcuts are chosen to be exclusively projections (1D convolutions with kernel size of 1) - at least in the original analysis, those were proven to be even more effective than identity connections, ([8, pg. 6]), but for this use case more experiments are needed to draw the same conclusion. At the same time, the same residual block structure was duplicated and an additional 1D convolutional layer was added, to have an analogous version with bigger receptive field (note that because of being 1D and not 2D (or more), the argument of parameter reduction from [11] does not apply here, only the one regarding nonlinearities). The two residual blocks described above were placed in parallel, and their results are simply concatenated when exiting the block; this would hopefully allow the model to "decide" in what measure it "needs" the layers with less receptive field or the layers with bigger receptive field (and also deeper).

Overall, the model's structure (which is depicted in figure 2) contains at first 3 inc1d blocks, then a layer which is implemented as a *foldl with sum operation*, then two LSTM layers. Using a combination of convolutions and then RNN-related structures for this kind of task has benn previously documented in [3], in which the authors stacked multiple convolutional layers with GRU layers - the intuition that was presented was that the convolutions help to create a more abstract and meaningful representation of the keystroke timeslice, and then the RNN structure processes it as a time sequence. Indeed, at least for the above mentioned paper and the current one, this combination was proven to be effective (a comparison with/without LSTM layers is presented in section 4). The use of LSTM instead of GRU layers in the number of 2 layers, in combination with batch normalization, is inspired by the constructon of TypeNet [2] - one discrepancy between the RNN structure used is the absence of the dropout layer. Initially, a dropout of 0.2 was added between the first and the second LSTM layer, and another after the second

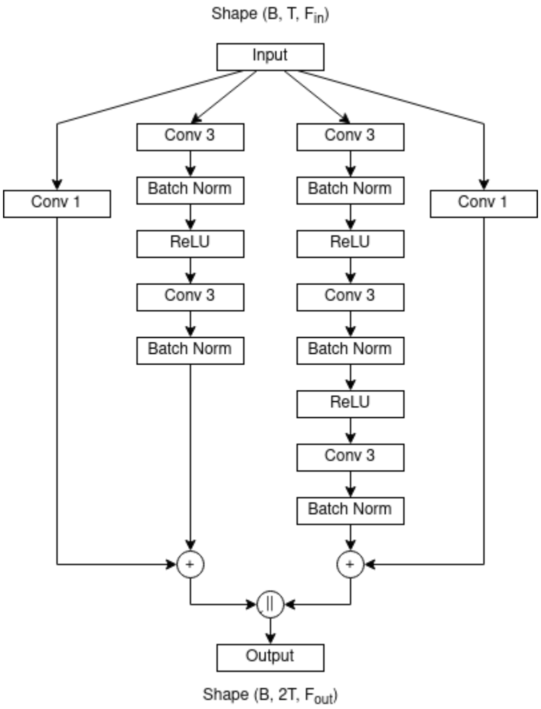

Figure 1: Custom block (*inc1d*) structure (conv parameter indicates kernel size)

layer. Although this construction yielded in the end about the same performance as the model without dropout, the validation loss value during the encoder train phase was showing a rather unstable and noisy behaviour (compared with the version without dropout), so it was ultimately removed (moreover, [12, section 3 a)] seems to suggest directly chaining dropout and batch normalization will have a negative impact).

The folded sum was introduced between the inc1d convolution-based layers and LSTM layers because of the concatenation operation that is present inside the inc1d block that acts on the time axis: an input has the shape $(B, T, 1)$, where $B$ is the batch size, $T$ is the timeslice len, the last "1" indicates the feature channel, initially containing only the key press timestamps relative to the beginning of the slice, as detailed in section 2. When passing through an inc1d block, the input shape $(B, T, F_{in})$ is converted in $(B, 2*T, F_{out})$, so after the 3 inc1d blocks that are used, the final shape is $(B, 8*T, F_{final})$, output which needs to be fed into LSTM layers. To reduce the time dimension to the original size, the output is folded over on the time dimension, with the step equal to the timeslice len $T$, such that the input for the LSTM layers has the shape $(B, T, F_{final})$. An alternative approach would have been to concatenate on the feature axis instead of time axis in the inc1d block, and either fold over the feature axis or not fold at all and feed the output to the LSTM layers as-is - this has not been tested, though.

The LSTM layer reduces the time dimension and outputs a vector of shape $(B, F_{final})$. This output is then passed to a dense layer with 128 neurons with ReLU activation, which is attached to the encoder and is responsible for creating the embedding vector for each timeslice in the batch. After the encoder training phase has ended, the last embedding layer will be discarded, and the encoder is then attached to the classifier.

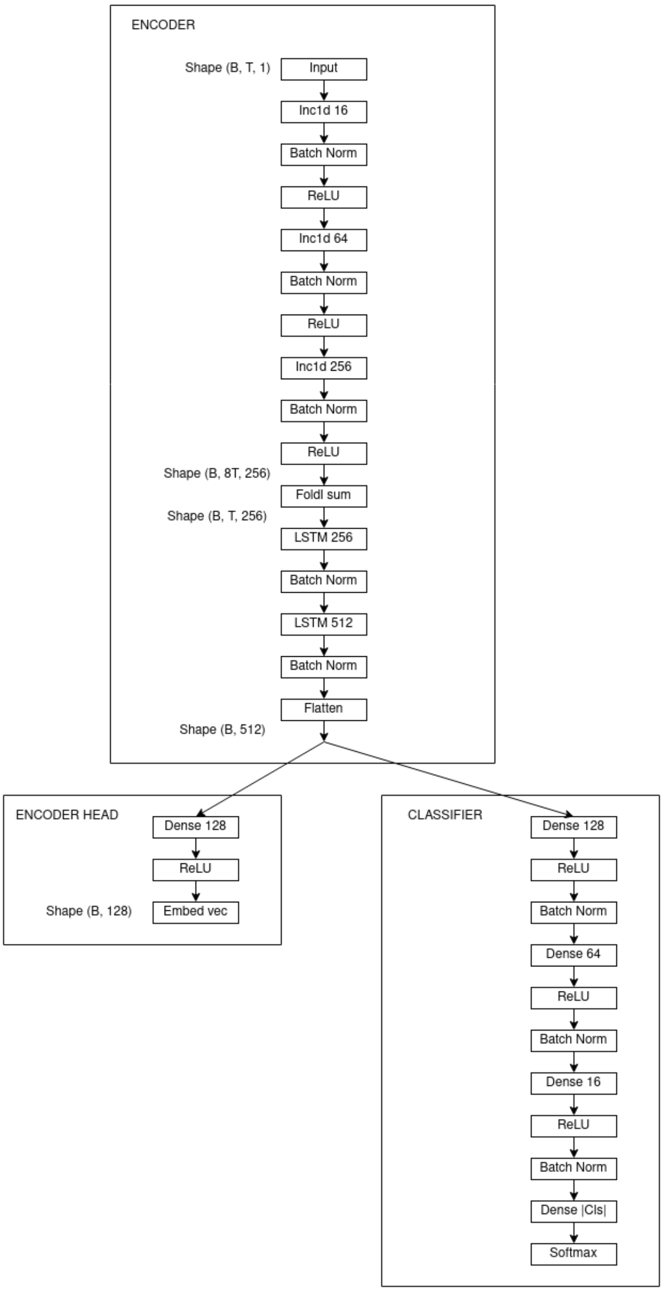

Figure 2: Neural network architecture

### 3.1.2 Classifier

The second component is the *classifier*, which is attached to the encoder and then it is trained with crossentropy loss, while the encoder's weights remain unchanged. The structure of the classifier can be visualised along the rest of the model in figure 2. It consists of 4 dense layers with 128, 64, 16, and $|class|$ units (in this particular case, $|class|$ is equal to 2).

## 3.2 Other variations

# 4 Results

In this section the results of training and validation for multiple model variations are presented, which, unless explicitly stated, share the same characteristics with the base model:

- the base model
- the base model without contrastive learning (classifier and encoder are trained at the same time)
- the base model without LSTM layers
- the base model with no overlapping time sequences in the train data
- the base model with timeslices of length 16 (instead of 30)
- the base model with residual block instead of the custom inc1d block

It should be acknowledged from the beginning though, that the results were generated using the same seed (123), and that the results that use other RNGs vary by a rather large amount - for example, for the base model, accuracies between 74% and 88% were found. This might be partially due to a small number of validation samples ( 280 samples were used - 140 from each class, the ratio between train:validation being 9:1). Another factor that needs to be taken into account is the context in which the data has been collected; as mentioned in section 2, it is primarily collected from typed python code and natural language text. The aim of this experiment is to test the performance of the model in a *free-text* scenario, so additional restrictions on the data would inevitably shift the current scenario towards a *fixed-text* situation.

Regarding the seeds used, they control the RNGs used by the native random module, numpy, and tensorflow, and thus any ordering of data inside the python code can be fully replicated. Still, in the train phase itself, other random sources associated with GPU processing come into play. This means that, even when the same seed is used, there will be slight fluctuations from the results presented.

Another observation would be that the performance is measured in *accuracy.* Other statistics (i.e. TAR, FAR, EER) that are usually associated with biometric authentication methods are not used, again, due to the limited data available.

The optimizers, the loss functions, batch sizes, and the number of epochs, unless otherwise specified, are the following:

- The encoder was trained for 100 epochs, and the classifer for 60 epochs.
- Batch size 16.
- The optimizer for both encoder train and classifier train was SGD with $10^{-4}$ learning rate and 0.9 momentum (as provided by Keras framework).
- The loss function for the encoder was SupCon ([7], [13]) with temperature $\tau = 0.1$. The loss function for the classifier was categorical cross-entropy (as provided by Keras framework).

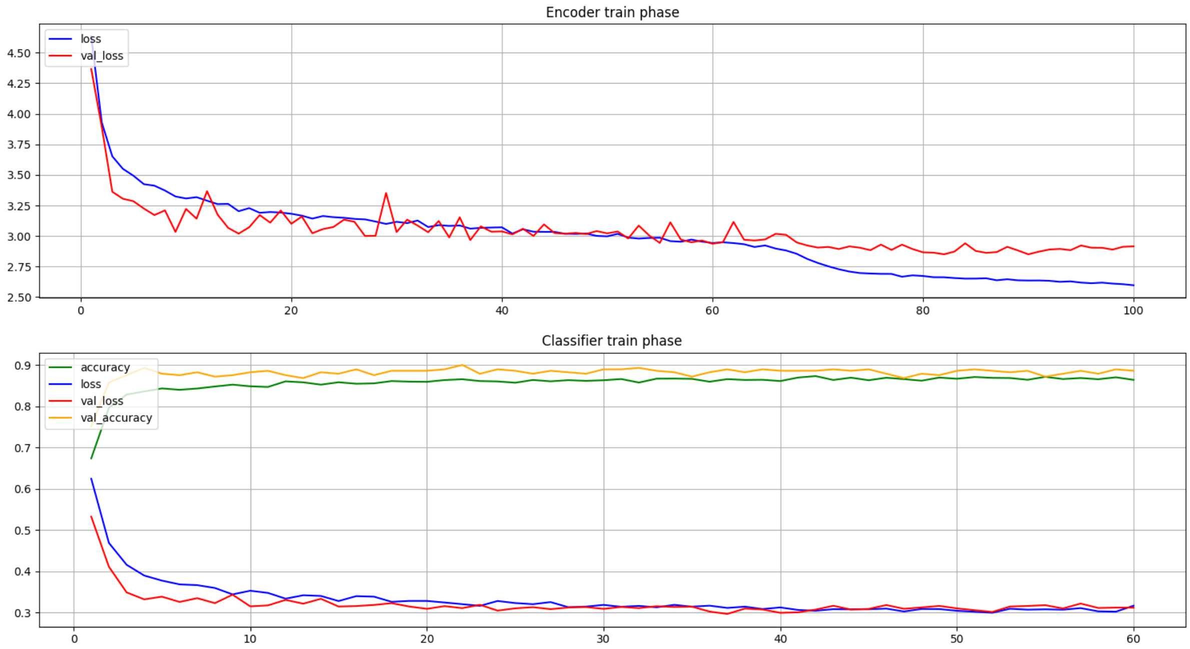

Figure 3: Base model with seed 123

## 4.1 Base model

In figure 3 the model yielded 88% validation accuracy. That being said, it can be observed that the validation accuracy is *higher* than the train accuracy. This might further indicate that the data, at least up to some point, is heterogeneous in features even if it belongs to the same class (person), and the RNG initialization with that seed simply happened to select examples for the validation set that are easier to classify than others. Another run with another seed can be observed in figure 4 with 81% accuracy, where the previous effect is not observed anymore. Still, the validation accuracy appears to be substantially noisier and less corelated with train accuracy.

Other runs revealed, as stated before, fluctuations in validation accuracy as low as 74%, but with loss curves (at least in the encoder training phase) having the same trend.

## 4.2 Model without overlapping timeslices

This comparison was done to directly test the efficiency of using (partially) overlapping timeslices, as presented in section 2. Without the overlapping timeslices, the train timeslice count was only 1305 * 2, and the validation count was 144 * 2 (the validation data did not contain overlapping sequences anyways). The results can be seen in figure 5, having 74.5% validation accuracy. Note that while the seed is the same as for the figure 3, the fact that the timeslices differ makes it harder to compare both results. Still, it yielded an accuracy which is lower than most other results when the overlapping timeslices were taken into account.

## 4.3 Model with timeslice length of 16

This other variation (that is related to the one before) was expected to yield lower results because the smaller timeslice length limits the correlations that can be detected by the model on the time axis - on

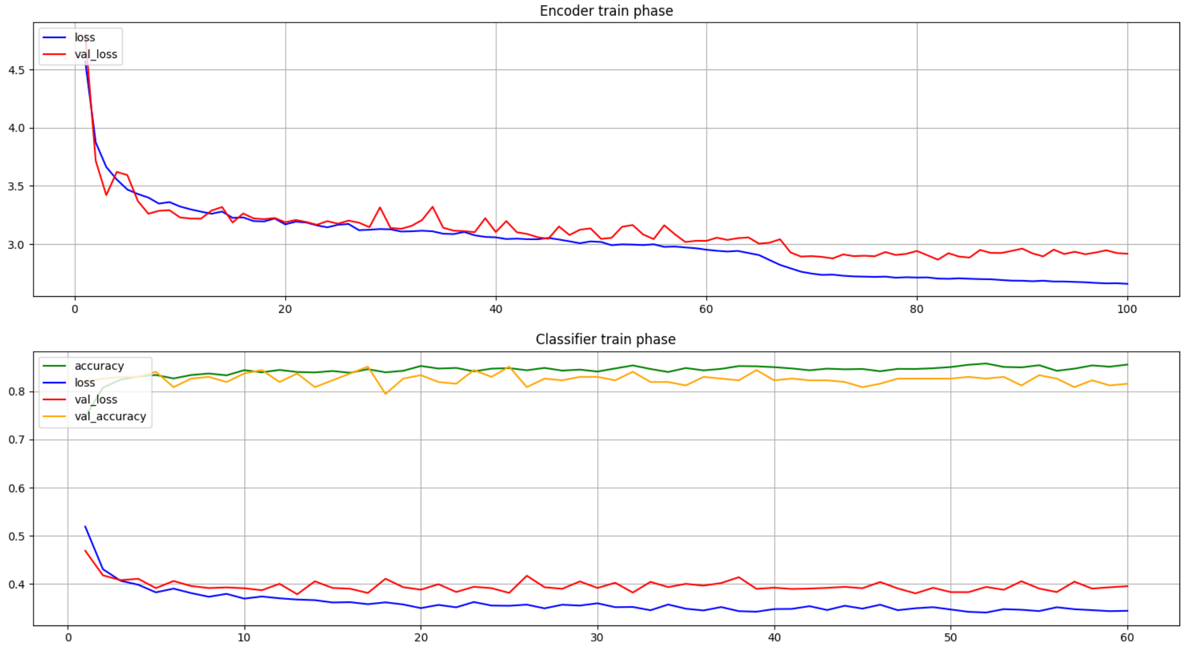

Figure 4: Base model with seed 55556

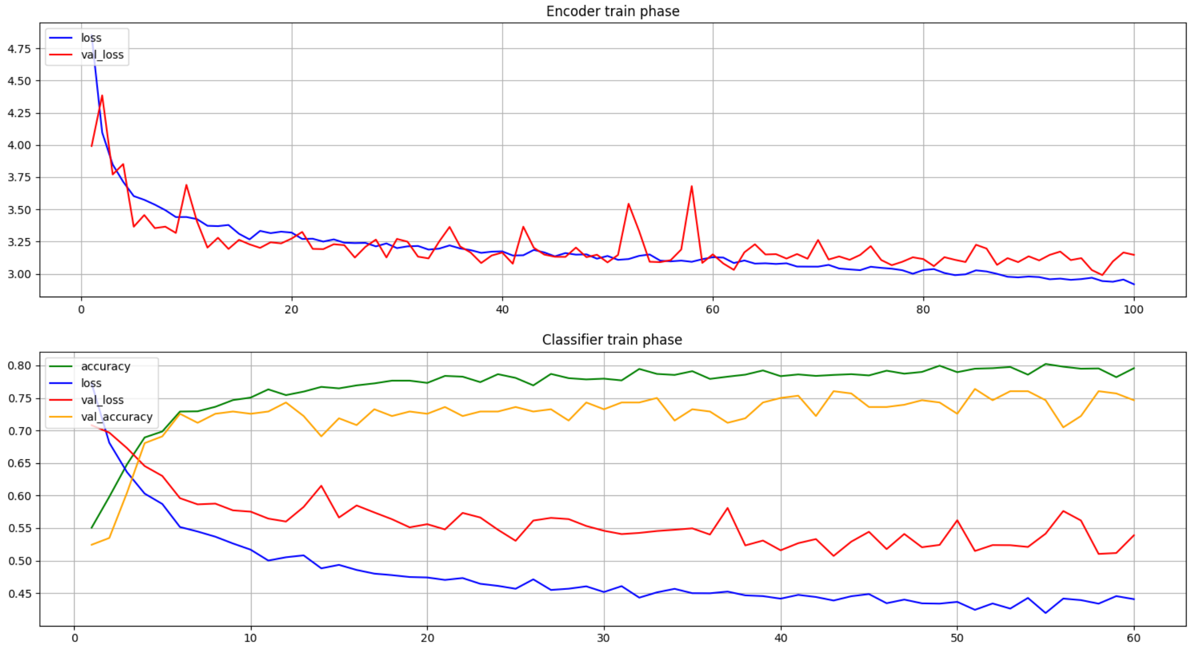

Figure 5: Base model without overlapping timeslices with seed 123

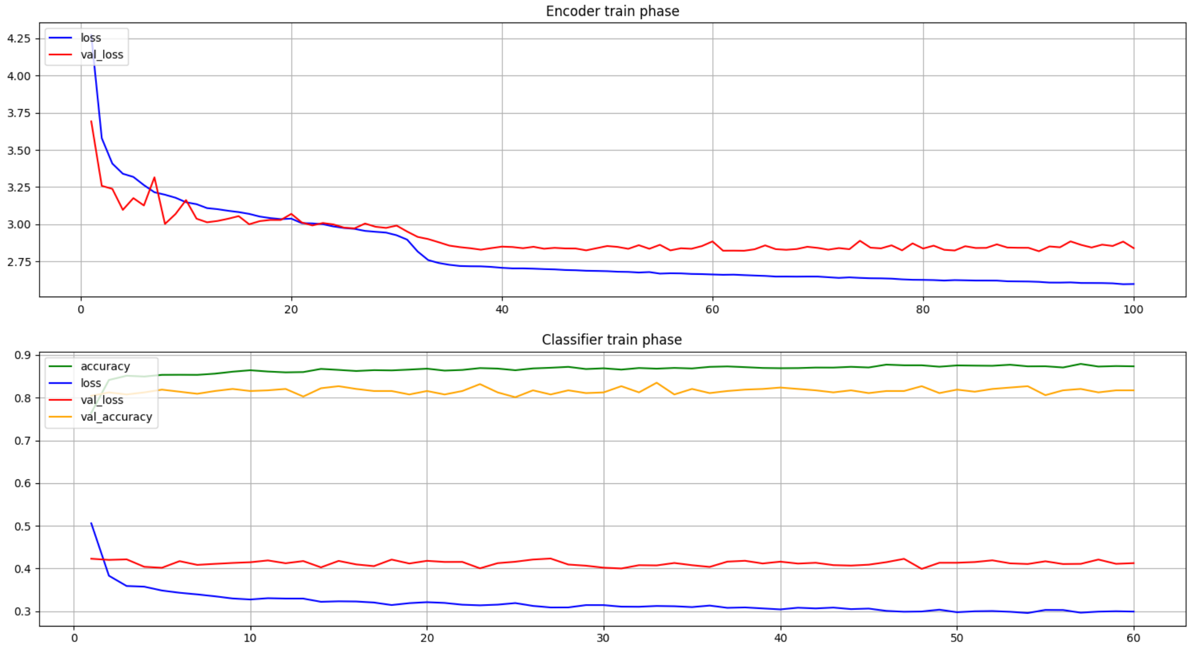

Figure 6: Base model using timeslices of length 16, with seed 123

the other side, splitting into smaller timeslices means their count grows (5207 * 2 timeslices for train, out of which 2424 * 2 overlapping, and 309 * 2 validation timeslices, none overlapping), and if most of the patterns detected by the model were lying in a small timestamp range (up to 16 in this case), the accuracy would be expected to grow. Indeed, the stats obtained from a few runs indicate the model apprears slightly affected by the halving of the timeslice length, but it still shows significant accuracy. it can be observed in figure 6 that one of the rounds obtained a validation accuracy of 82%; another thing to observe though, is that the 60 classifier training epochs were not needed, and even 2-3 epochs were enough to yield the same result.

## 4.4 Model without contrastive learning

The model without contrastive learning has the exact same structure as the encoder + classifier from previous contrastive learning, but they are fused into one single neural net from the beginning and trained in one single phase for 150 epochs. One can immediately see from the figure 7 that this model simply fails to converge. Although the validation accuracy after the last epoch was 88%, this is simply a coicidence: stopping the training a few epochs earlier would have shown an accuracy of around 65%, which, given the small number of validation samples, is only slightly better than guessing. This is a strong indication that, at least for this task, contrastive learning (or at least contrastive loss functions) significantly improve performance - this is also empirically proven in [2].

## 4.5 Model without LSTM layers

This is one of the most interesting analysed cases: a small experiment was done, in which the overall model was trained 2 times, with one single difference: the first time the encoder was trained for 100 epochs (figure 8), and in the second run the encoder was trained for only 40 epochs (figure 9). Both models show an overfit on the encoder training loss curves, but after the classification training phase,

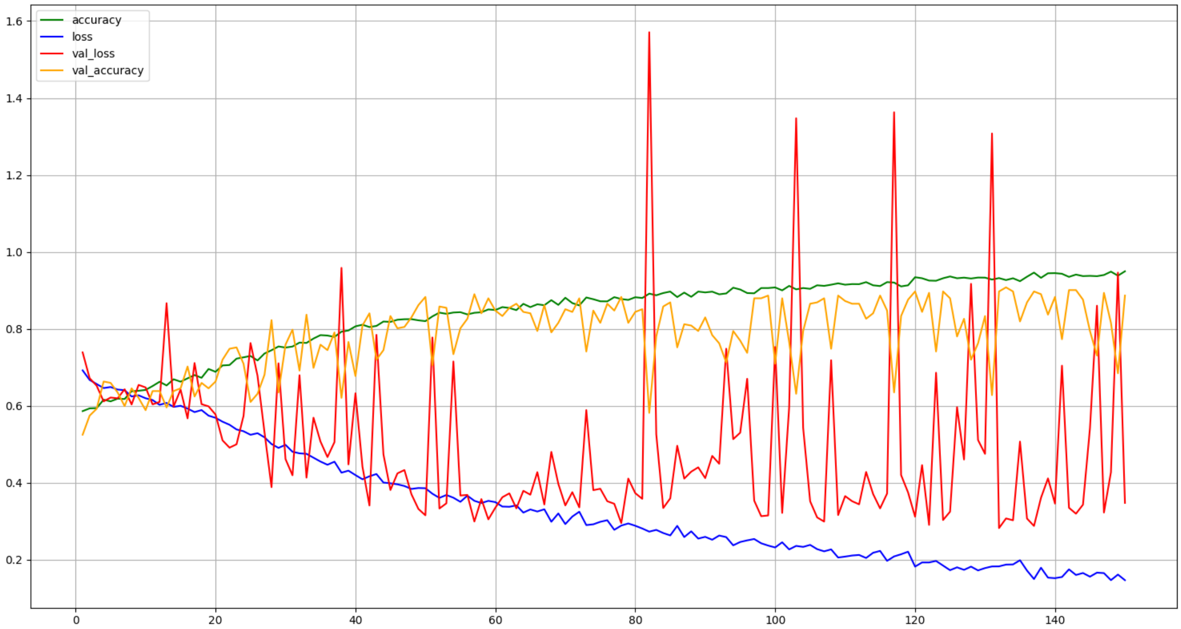

Figure 7: Model without contrastive learning, with seed 123

the model which appears to be overfitted the most (the one with 100 encoder train epochs) actually performes better (accuracy 84% vs accuracy 74%). One other aspect is that, in both cases, the encoder validation loss *is at least as big as it was after the first epoch*; still, after the classifier train phase, the model is still able to predict with decent accuracy. It is worth noting that, while investigating different model structures, this counterintuitive situation appeared multiple times - what fixed it, is the addition of LSTM layers. All of this sugests that the CNN+RNN combination first tested in [3] is proven to help create a more meaningful representation of the dataset, thus stabilizing the model and increasing its accuracy (RNN-only models resembling TypeNet in [2] were tested, but without much success - they were not included in this documentation).

### 4.6 Model with residual blocks instead of inc1d

This comparison tests whether there are any improvements the custom, residual-based inc1d blocks bring over their original residual blocks counterparts. The structure of the residual block used is exactly the same as the left branch depicted in the figure 1 which describes the inc1d block. Some runs of this version suggests slight improvements of inc1d over simple residual blocks (i.e. they almost never surpass the 80% threshold), although this model also shows stability, at least compared to some other variations; it yielded 75% accuracy, as shown in figure 10.

## 5 Conclusion

Various deep learning models were tested as a proof-of-concept versions for discriminating between different people, taking into account only keystroke press timestamps. The data used helped in proving that the above mentioned task is achievable in (at least some) free-text scenarios, with a minimal number of features and information collected from the environment. The model architecture used a

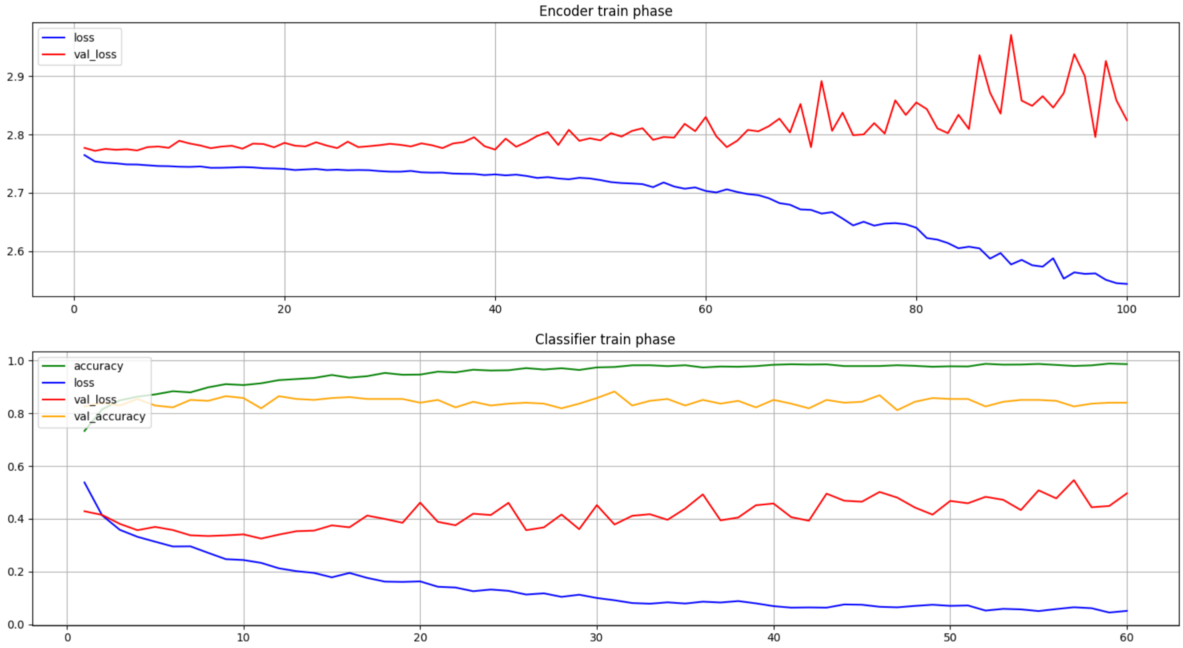

Figure 8: Model without LSTM layers, 100 encoder epochs, seed 123

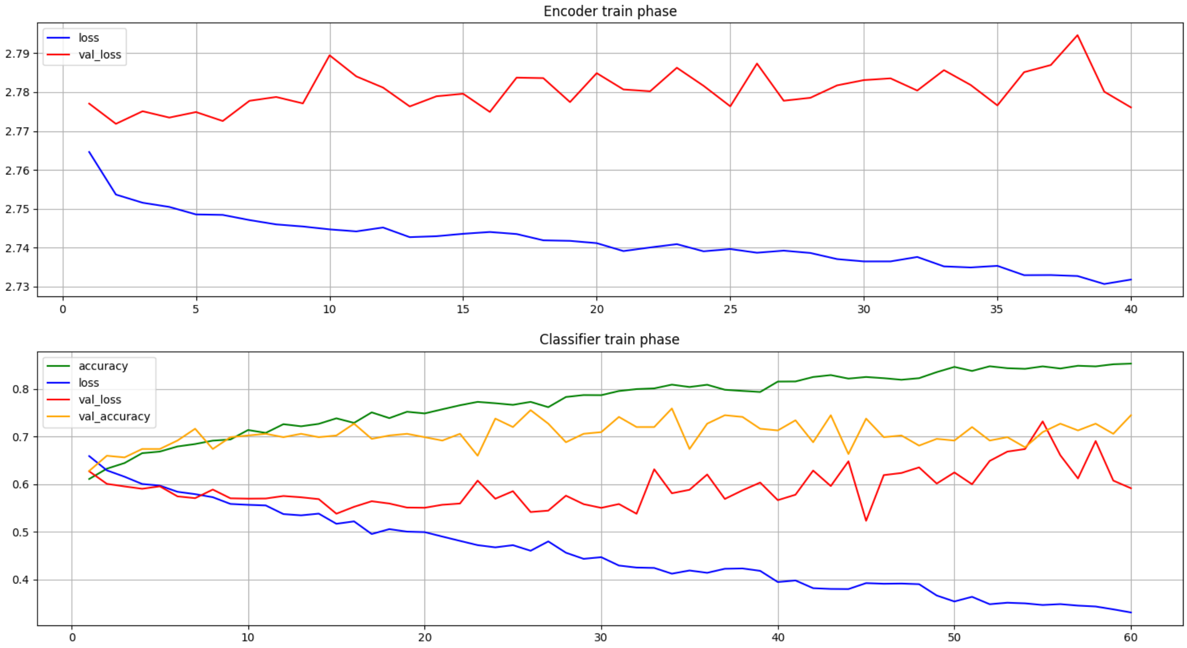

Figure 9: Model without LSTM layers, 40 encoder epochs, seed 123

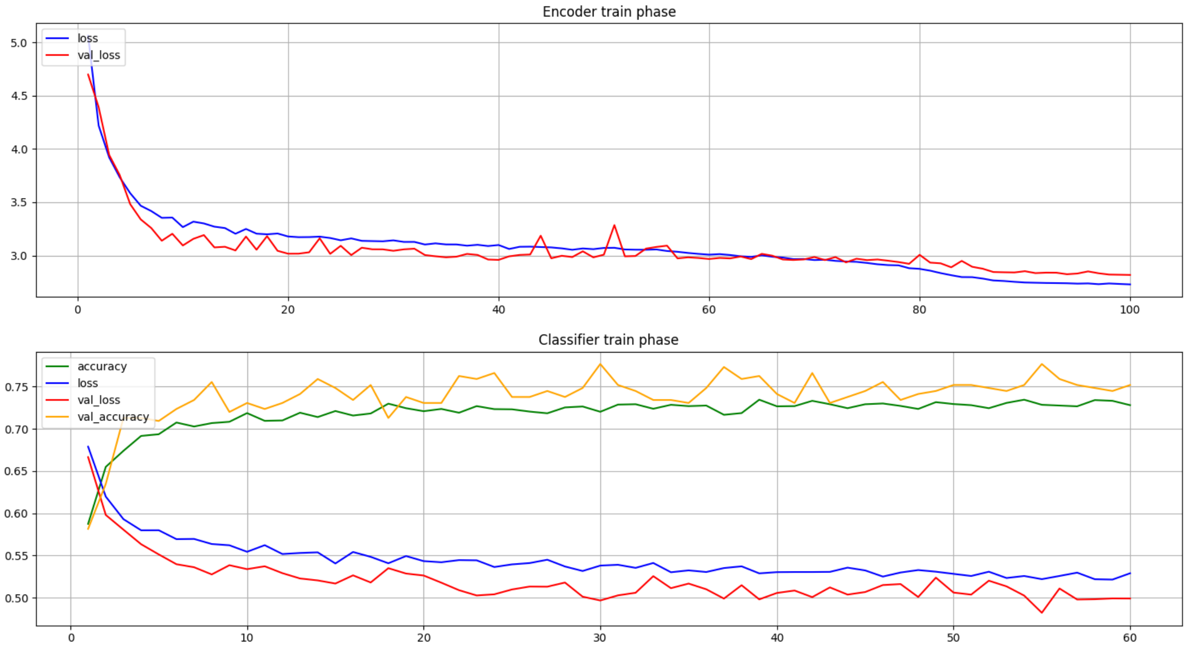

Figure 10: Model with residual blocks instead of inc1d blocks, seed 123

combination of convolution-based modules and recurrent layers, along with the supervised contrastive learning paradigm, and showing the efficiency of some of the features used, compared to alternatives. That being said, there is room for a lot of improvements, and it remains to be seen whether these conclusions can be generalised on a larger dataset, with potentially hundreds or even thousands of different classes.

## 6 Acknowledgements

I want to thank Ioan Savu and Sumanaru Andrei for helping me collect the data needed to run the entire project.

This document was written in the context of a computer vision / deep learning course at the University of Bucharest, held by a team from Arnia Software.

## 7 Additional note

As with the ”random vs human” classification task mentioned in section 2, and all the other customizable implementation features, this implementation also includes a *genetic algorithm* implementation that has the role to evolve a small model. This feature has been used early on, to help improve a very small convolutional network to distinguish between random data and human data. While this performance is easily achievable with a deeper network / more epochs, and while this scenario is the easiest for a model to solve, I still consider it interesting to have another algorithm bring the performance of some very small random network up to 97% accuracy by itself.

The genetic algorithm encodes a small convolutional network architecture in a bit array (expressed as an integer for convenience). Slices of that chromosome encode, separately, properties of layers (eg.

number of channels in a conv layer), or even if the layer is used at all or not. The genetic operators that were implemented were mutation (with a specified per-bit probability) and crossover (again, with specified probability). The explosion of possibilities was controlled by limiting the nature of the network that can be ”born” (for example, the conv layers always were the first, then the dense layers), and also by the fact that the number of training epochs was very small (up to 10-20, each of them lasting at most 2-3 seconds). For the curious reader, all the implementation details can be found in the file *evol.py* attached with the rest of the source code.

## References

[1] ZHONG, Yu; DENG, Yunbin. *A survey on keystroke dynamics biometrics: approaches, advances, and evaluations.* Recent Advances in User Authentication Using Keystroke Dynamics Biometrics, 2015, 1: 1-22.

[2] ACIEN, Alejandro, et al. *TypeNet: Deep learning keystroke biometrics.* IEEE Transactions on Biometrics, Behavior, and Identity Science, 2021.

[3] XIAOFENG, Lu; SHENGFEI, Zhang; SHENGWEI, Yi. *Continuous authentication by free-text keystroke based on CNN plus RNN.* Procedia computer science, 2019, 147: 314-318.

[4] CHO, Kyunghyun, et al. *Learning phrase representations using RNN encoder-decoder for statistical machine translation.* arXiv preprint arXiv:1406.1078, 2014.

[5] HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. *Long short-term memory.* Neural computation, 1997, 9.8: 1735-1780.

[6] MURPHY, Christopher, et al. *Shared dataset on natural human-computer interaction to support continuous authentication research.* In: 2017 IEEE International Joint Conference on Biometrics (IJCB). IEEE, 2017. p. 525-530.

[7] KHOSLA, Prannay, et al. *Supervised contrastive learning.* Advances in Neural Information Processing Systems, 2020, 33: 18661-18673.

[8] HE, Kaiming, et al. *Deep residual learning for image recognition.* In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. p. 770-778.

[9] SZEGEDY, Christian, et al. *Going deeper with convolutions.* In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2015. p. 1-9.

[10] GERS, Felix A.; SCHMIDHUBER, Jürgen; CUMMINS, Fred. *Learning to forget: Continual prediction with LSTM.* Neural computation, 2000, 12.10: 2451-2471.

[11] SIMONYAN, Karen; ZISSERMAN, Andrew. *Very deep convolutional networks for large-scale image recognition.* arXiv preprint arXiv:1409.1556, 2014.

[12] LI, Xiang, et al. *Understanding the disharmony between dropout and batch normalization by variance shift.* In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019. p. 2682-2690.

[13] https://keras.io/examples/vision/supervised-contrastive-learning/

[14] https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM

[15] https://github.com/mousepad01/nn_on_palinca